Automated Timed Temporal Verification for a Mixed Sync-Async Concurrency Paradigm

ANONYMOUS AUTHOR(S)

To make reactive programming more concise and flexible, it is promising to deploy a mixed concurrency paradigm [Berry and Serrano 2020] that integrates Esterel's synchrony and preemption [Berry 1999] with JavaScript's asynchrony [Madsen et al. 2017]. Existing temporal verification techniques have not been designed to handle such a blending of two concurrency models. We propose a novel solution via a compositional Hoare-style forward verifier and a term rewriting system (TRS) on *Timed Synchronous Effects* (TSE).

Firstly, we introduce TSE, a new effects logic, that extends *Kleene Algebra* with value-dependent constraints, providing real-time bounds for logical-time synchronous traces [Von Hanxleden et al. 2017]. Secondly, we establish an axiomatic semantics for a core language λ_{HH}, generalising the mixed paradigm. Thirdly, we present a purely algebraic TRS, to efficiently check language inclusions between expressive timed effects. To demonstrate the feasibility of our proposals, we prototype the verification system; prove its correctness; investigate how it can help to debug errors related to both synchronous and asynchronous programs.

Additional Key Words and Phrases: Temporal Verification, Dependant Effects, Hoare-style Forward Verifier, Term Rewriting System, Reactive System Design

1 INTRODUCTION

As it recently proposed, reactive languages such as Hiphop.js[1] [Berry and Serrano 2020; Vidal et al. 2018] are designed to be based on a smooth integration of (i) Asynchronous concurrent programs, which perform interactions between components or with the environment with uncontrollable timing, such as network-based communication; (ii) Synchronous reactive programs, which react to external events in a conceptually instantaneous and deterministic way; and (iii) Preemption, the explicit cancellation and resumption of an ongoing orchestration subactivity.

"Such a combination makes reactive programming more powerful and flexible than plain JavaScript because it makes the temporal geometry of complex executions explicit instead of hidden in implicit relations between state variables." [Berry and Serrano 2020]

Given such a multi-paradigm (mixed Sync-Async) concurrency model, the verification of its temporal behaviour becomes engaging and challenging. Existing techniques are based on a transformation to convert the asynchronous chunk into semantically equivalent synchronization[2]; then reason about the behaviours based on the synchronous semantics [Damian et al. 2019; Gleissenthall et al. 2019; Tarawneh and Mokhov 2018]. This approach (i) suffers from the limited expressiveness, restricted by finite-state automata (FSA); (ii) lacks the modularity to reason about programs compositionally; (iii) loses time awareness due to the conversion to synchronous models; and (iv) has the *state explosion problem* when proving language inclusions between FSA.

To tackle the above issues and exploit the best of both synchronous and asynchronous concurrency models, we propose a novel temporal specification language, which enables a compositional verification via a Hoare-style forward verifier and a term rewriting system (TRS). More specifically, we specify system behaviours in the form of *Timed Synchronous Effects (TSE)*, which integrates the Synchronous Kleene Algebra (SKA) [Broda et al. 2015; Prisacariu 2010] with dependent values and

[1] Hiphop.js is a JavaScript extension of Esterel [Berry and Gonthier 1992] (or vice versa) for reactive web applications: https://www-sop.inria.fr/members/Colin.Vidal/hiphop/.

[2] Usually the transformation is incomplete, i.e., not all the asynchronous programs can be translated into a semantically equivalent synchronization.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

arithmetic constraints, to provide real-time abstractions into traditional synchronous verification. For example, one safety property, *"The event* **Done** *will be triggered no later than one second"*[3], is expressed in our effects logic as:

$$\Phi \triangleq 0{\leq}t{<}1000 : (\{\}^{\star} \cdot \{\textbf{Done}\})\#t.$$

Here, # is the parallel operator specifying the *real-time* constraints for the *logical-time* sequences [Von Hanxleden et al. 2017]; curly braces {} enclose a single logical-time instance[4]; Kleene star $\star$ denotes trace repetition. The above formula Φ corresponds to '$\Diamond_{[0,1000)}$ **Done**' in metric temporal logic (MTL), reads *"within one second,* **Done** *finally happens"*. Moreover, the time bounds can be dependent on the program inputs. For example, we express, the effects of $send(d)$ as:

$$\Phi^{send(d)} \triangleq (0{<}d{\leq}5000 \wedge 0{\leq}t{<}d) : (\{\textbf{Send}\}\#t) \cdot \{\textbf{Done}\}.$$

The *send* method takes a parameter d, and **Send**s out a message at time t. The above formula $\Phi^{send(d)}$ indicates that the input d is positive and smaller or equal to *5000*; the method generates a finite trace, with the first time instance containing the event **Send**, followed by a second time instance containing the event **Done**; and the instance {**Send**} takes a no more than d milliseconds delay to finish. Although these examples are simple, they show the benefits of deploying value-dependent time bounds. Intuitively, if traditional timed automata [Larsen et al. 1997] define an *exact* transition system, our timed effects define a set (possibly infinite) of exact transition systems.

Having the effects logic as the specification language, we are interested in the following verification problem: Given a program $\mathcal{P}$, and a temporal property Φ', does $\Phi^{\mathcal{P}} \sqsubseteq \Phi'$ holds? In a typical verification context, checking the inclusion/entailment between the program effects $\Phi^{\mathcal{P}}$ and the valid traces Φ' proves that: the program $\mathcal{P}$ will never lead to unsafe traces which violate Φ'.

Traditional ways of temporal verification either (i) rely on a translation from specification languages, such as LTL or CSP, into finite state automata [Sun et al. 2009], which potentially gives rise to an exponential blow-up; or (ii) use expressive automata to model the program logic directly, such as timed automata, which fails to capture the bugs introduced by the real implementation.

In this paper, we present a new solution of extensive temporal verification comprising: a front-end verifier computes the program's temporal behaviour, to be the $\Phi^{\mathcal{P}}$, via inference rules at the source level; and a back-end TRS, to soundly check $\Phi^{\mathcal{P}} \sqsubseteq \Phi'$.

The TRS is inspired by Antimirov and Mosses' algorithm [Antimirov and Mosses 1995][5] but solving the language inclusions between more expressive timed synchronous effects. A TRS is a refutation method that normalizes expressions in such a way that checking their inclusion corresponds to an iterated process of checking the inclusion of their *partial derivatives* [Antimirov 1995]. Works based on such a TRS [Almeida et al. 2009; Antimirov and Mosses 1995; Hovland 2012; Keil and Thiemann 2014; Song and Chin 2020] show its feasibility and suggest that this method is a better average-case algorithm than those based on the comparison of automata.

This work targets timed temporal verification, and to the best of the authors' knowledge, it proposes the first axiomatic semantics and the first algebraic TRS (proven sound and complete) for the mixed synchronous and asynchronous concurrency model. Our main contributions are:

(1) **The Timed Synchronous Effects:** We define the syntax (Sec. 4.2) and semantics (Sec. 4.3) of timed synchronous effects, to be the specification language, which captures the target programs' behaviours and non-trivial temporal properties.

[3]For simplicity, we follow the convention of JavaScript, and use integer values to represent milliseconds in this paper, while it can be easily extended to real numbers and other time measurement units.

[4]A time instance is a set of signals (possible empty) with status, logically concurring at the same time (cf. Sec. 4.2).

[5]Antimirov and Mosses' algorithm was designed for deciding the inequalities of regular expressions based on an complete axiomatic algorithm of the algebra of regular sets.

(2) **Automated Forward Verifier:** Targeting a core language λ_{HH} (Sec. 4.1), we establish its axiomatic semantics via a set of inductive transition rules (Sec. 5), enabling a compositional verifier to infer the program's effects. The verifier triggers the back-end solver TRS.
(3) **An Efficient TRS:** We present the rewriting rules, to prove the inferred effects against given temporal properties, both expressed by timed synchronous effects (Sec. 6).
(4) **Implementation and Evaluation:** We prototype the novel effects logic and the automated verification system, prove the correctness, report on a case study investigating how it can help to debug errors related to both synchronous and asynchronous programs (Sec. 7.2).

Organization. Sec. 2 introduces the language features of synchronous Esterel programs, JavaScript promises, and the web reactive language HipHop.js. Sec. 3 gives motivation examples to highlight the key methodologies and contributions. Sec. 4 formally presents the core language λ_{HH}, and the syntax and semantics of timed synchronous effects. Sec. 5 presents the forward verification rules. Sec. 6 explains the TRS for effects inclusion checking, and displays the essential auxiliary functions. Sec. 7 demonstrates the implementation and cases studies. We discuss related works in Sec. 8 and conclude in Sec. 9. Omitted proofs can be found in the Appendix.

2 BACKGROUND: ESTEREL, ASYNC-AWAIT, AND HIPHOP.JS

It has been an active research topic to build flexible programming paradigms for reactive systems in different domains. This section identifies: i) *Synchronous programming*, represented by Esterel [Berry and Gonthier 1992], ii) *Asynchronous programming*, represented by JavaScript promises [Madsen et al. 2017], and iii) A *mixed Sync-Async* paradigm, recently proposed by HipHop.js [Berry and Serrano 2020]. Meanwhile, we discuss the real-world programming challenges of each paradigm; and show how our proposal addresses them in the cases studies, Sec. 7.2.

2.1 A Sense of Esterel: Synchronous and Preemptive

The principle of synchronous programming is to design a high-level abstraction where the timing characteristics of the electronic transistors are neglected [est 2021]. Such an abstraction makes reasoning about time a lot simpler, thanks to the notion of *logical ticks*: a synchronous program reacts to its environment in a sequence of ticks, and computations within a tick are assumed to be instantaneous. As one of the first few well-known synchronous languages, Esterel's high-level imperative style allows the simple expression of parallelism and preemption, making it natural for programmers to specify and reason about control-dominated model designs. Esterel has found success in many safety-critical applications such as nuclear power plant control software. The success with real-time and embedded systems in domains that need strong guarantees can be partially attributed to its precise semantics and computational model [Berry 1999; Berry and Gonthier 1992; Song and Chin 2021].

Esterel treats computation as a series of deterministic reactions to external signals. All parts of a reaction complete in a single, discrete-time step called an *instance*. Besides, instances exhibit deterministic concurrency; each reaction may contain concurrent threads without execution order affecting the computation result. Primitive constructs execute in zero time except for the *yield* (pause) statement. Hence, time flows as a sequence of logical instances separated by explicit pauses. In each instance, several elementary instantaneous computations take place simultaneously.

```
1  fork {emit A; yield; emit B; emit C} par {emit E; yield; emit F; yield; emit G}
```

The synchronous parallelism in Esterel is constructed by the *fork*{...}*par*{...} statement. It remains active as long as one of its branches remains active, and it terminates when both branches are terminated. The branches can terminate in different instances, and wait for the last one to

terminate. As the above example shows, the first branch generates effects $\{\mathbf{A}\} \cdot \{\mathbf{B}, \mathbf{C}\}$ while the second branch generates effects $\{\mathbf{E}\} \cdot \{\mathbf{F}\} \cdot \{\mathbf{G}\}$; then the final effects should be $\{\mathbf{A}, \mathbf{E}\} \cdot \{\mathbf{B}, \mathbf{C}, \mathbf{F}\} \cdot \{\mathbf{G}\}$.

To maintain determinism and synchrony, evaluation in one thread of execution may affect code arbitrarily far away in the program. In other words, there is a strong relationship between signal status and control propagation: a signal status determines which branch of a *present* test is executed, which in turn determines which *emit* statements are executed (See Sec. 4.1 for the language syntax). The first semantic challenge of programming Esterel is the *Logical Correctness* issue, caused by these non-local executions, which is simply the requirement that there exists precisely **one** status for each signal. For example, consider the program below:

```
signal S1 in present S1 then nothing else emit S1 end present end signal
```

If the local signal **S1** were *present*, the program would take the first branch of the condition, and the program would terminate without having emitted **S1** (*nothing* leaves **S1** with *absent*). If **S1** were absent, the program would choose the second branch and emit the signal. Both executions lead to a contradiction. Therefore there are no valid assignments of signals in this program. This program is logically incorrect.

```
signal S1 in present S1 then emit S1 else nothing end present end signal
```

Consider the revised program above. If the local signal **S1** were present, the conditional would take the first branch, and **S1** would be emitted, justifying the choice of signal value. If **S1** were absent, the signal would not be emitted, and the choice of absence is also justified. Thus there are two possible assignments to the signals in this program, which is also logically incorrect.

Esterel's instantaneous nature requires a special distinction when it comes to loop statements, which increases the difficulty of the effects invariants inference. As shown in Fig. 1., the program firstly emits signal **A**, then enters into a loop which emits signal **B** followed by a *yield* followed by emitting signal **C** at the end. The effects of it is $\{\mathbf{A}, \mathbf{B}\} \cdot \{\mathbf{B}, \mathbf{C}\} \cdot \{\mathbf{B}, \mathbf{C}\} \cdot \{\mathbf{B}, \mathbf{C}\}$..., which says that in the first time instance, signals **A** and **B** will be present, as there is no explicit yield between *emit* **A** and *emit* **B**; then for the following instances (in an infinite trace), signals **B** and **C** are present all the time, because after executing *emit* **C**, it immediately executes from the beginning of the loop, which is *emit* **B**.

```
1 module a_loop: output A,B,C;
2   emit A;
3   loop
4     emit B; yield; emit C
5   end loop
6 end module
```

Fig. 1. A Loop Example in Esterel [Song and Chin 2021].

Coordination in concurrent systems can result from information exchange, using messages circulating on channels with possible implied synchronization. In Esterel, it can also result from *process preemption* [Berry 1993], which is a more implicit control mechanism that consists in denying the right to work to a process, either permanently (abortion, cf. Fig. 4.) or temporarily (e.g. suspension). Preemption is particularly important in control-dominated reactive and real-time programming, where most of the works consist of handling interrupts and controlling computation.

While the flexibility they provide us, preemption primitives often with loose or complex semantics, making abstract reasoning difficult. Most existing languages offer a small set of preemption primitives, often insufficient to program reactive systems concisely.

2.2 Asynchrony from JavaScript Promises: Async–Await

"Who can wait quietly while the mud settles? Who can remain still until the moment of action?"
– Laozi, Tao Te Ching

A number of mainstream languages, such as C#, JavaScript, Rust, and Swift, have recently added support for async–await and the accompanying promises abstraction[6], also known as *futures* or *tasks* [Bierman et al. 2012]. As an example, consider the JavaScript program in Fig. 2., it uses the `fs` module (line 1) to load the file into a variable (line 6) using async/await syntax.

```
1  const fs = require('fs').promises;
2
3  async function read (filePath) {
4      const task = fs.readFile(filePath);
5      ... // do things that do not depend on the result of the loading file
6      const data = await task; // block execution until the file is loaded
7      ... // logging or data processing of the Json file
8  }
```

Fig. 2. Using Async-Await in JavaScript.

The function `read` accepts one argument, a string called `filePath`. As it is declared **`async`**, reading a file (line 4) does not block computations that do not depend on the result (line 5). The programmer **`await`**s the task when they need the result to be ready. Reading a file may raise exceptions (e.g., the file is non-existent), then the point for such an exception to emerge is where the tasks are awaited (lines 6). While await sends a signal to a task scheduler on the runtime stack, the exceptions appear to propagate in the opposite direction, from the runtime to the await sites.

Unfortunately, existing languages that support async–await do not enforce at compile time that exceptions raised by asynchronous computations are handled. The lack of this static assurance makes asynchronous programming error-prone. For example, the JavaScript compiler accepts the program above without requiring that an exception handler to be provided, which leads to program crashes if the asynchronous computation results in an exception. Such unhandled exceptions have been identified as a common vulnerability in JavaScript programs [Alimadadi et al. 2018; Madsen et al. 2017]. Furthermore, [Alimadadi et al. 2018; Madsen et al. 2017] display a set of other *anti-patterns* including: attempting to settle a promise multiple times; unsettled promises; unreachable reactions; and unnecessary promises, which are mostly caused by using promises without sufficient static checking.

2.3 HipHop.js – A mixture of Esterel and JavaScript

```
1  function enableLoginButton(){
2    return (Rname.length >= 2
3         && Rpasswd.length >= 2);}
4  function nameKeypress(value){
5    Rname = value;
6    RenableLogin=enableLoginButton();}
7  function passwdKeypress(value){
8    Rpasswd = value;
9    RenableLogin=enableLoginButton();}
```

```
1  hiphop module Identity(
2              in name, in passwd,
3              out enableLogin){
4    do{
5        emit enableLogin(
6                 name.length >= 2
7               && passwd.length >= 2);
8    } every(name || passwd)
9  }
```

Fig. 3. A comparison between JavaScript (left) and HipHop.js (right) for a same login button implementation [Berry and Serrano 2020]. (On the right, `name`, `passwd` and `enableLogin` are reactive input/output signals.)

[6]JavaScript's asynchrony arises in situations such as web-based user-interfaces, communicating with servers through HTTP requests, and non-blocking I/O.

HipHop.js is a reactive web language that adds synchronous concurrency and preemption to JavaScript, which is compiled into plain JavaScript and executes on runtime environments [Berry and Serrano 2020]. To show the advantages of such a mixture, Fig. 3. presents a comparison between JavaScript and HipHop.js to achieve the same login button. Here, `Rname`, `Rpasswd`, `RenableLogin` are global variables to model the application's states. Dis/En-abling login is done by setting `RenableLogin`. However, while more and more features get added to the specification, state variable interactions can lead to a large number of implicit and invisible global control states.

Whereas, HipHop.js simplifies and modularizes designs, and synchronous signaling makes it possible to instantly communicate between concurrent statements to exchange data and coordination signals. Second, powerful event-driven reactive preemption borrowed from Esterel finely controls the lifetime of the arbitrarily complex program statements they apply, instantly killing them when their control events occur. More examples are discussed in the following sections.

3 OVERVIEW

3.1 Timed Synchronous Effects

As shown in Fig. 4., we define Hoare-triple style specifications (enclosed in `/*@ ... @*/`) for each program, which leads to a compositional verification strategy, where static checking and timed temporal verification can be done locally.

```
1 hiphop module authenticate(var d, var name, var passwd, in Tick, out Connecting, out Connected)
2 /*@ requires d>3000 : {}^*.{Login} @*/
3 /*@ ensures (3000<t/\t<d : {Connecting}#t.{Connected}) \/ (t=d : {Connecting}#t) @*/
4 {
5    abort count(d, Tick) { // Abort the execution at d milliseconds
6       async Connected {
7          emit Connecting;
8          // Execute authenticateSvc after a 3000 milliseconds' delay
9          setTimeout (authenticateSvc(name, passwd).post().then( v => this.notify(v)), 3000);}}}
```

Fig. 4. Dependent Values for Time Bounds (the program is from [Berry and Serrano 2020]).

The `authenticate` module checks the validity of the identity at each click on login button. The operations of requesting the server are wrapped in an **`abort`** statement, which preempts the execution when the responding time goes beyond d (given signal `Tick` is present every second).

After emitting the signal **Connecting**, it emits **Connected** if the connection succeeds before the preemptive deadline; otherwise, no signal is emitted. The precondition $d{>}3000 : \{\}^\star \cdot \{Login\}$ requires that the input variable d is greater than 3000, and before entering into this module, the signal **Login** should be emitted at the last time instance, indicating that a login request has been sent. The postcondition contains two parts and connected using the disjunction mark $\vee$: when $3000{<}t{<}d$, the effects contains two time instances, $\{Connecting\}$ and $\{Connected\}$, while the first time instance finishes at time t, which is no later than d milliseconds; otherwise, the effects contains one time instance $\{Connecting\}$ which finishes at time d. *(See a demo from [Demo1 2021])*

As for another example, shown in Fig. 5., the module `main` spawns two threads running in parallel. The first thread firstly waits for the signal **Ready**, then emits **Go**. The second thread firstly emits **Prep**, then calls the function `cook` asynchronously. The precondition of `main` requires no arithmetic constraints on its input values (expressed as *true*), neither any pre-traces (expressed as *emp* or ϵ, indicating an empty trace). The postcondition ensures a trace, where the second time instance contains at least one signal `Cook` and finishes within 3000 milliseconds after the completion

of the first time instance. Then we do not care about the rest of the trace ({} is similar to a wildcard). Taking `main` as an example, the work flow of the forward verifier is presented in the following sub-section. *(See a demo from [Demo2 2021])*

```
1  hiphop module main (out Prep, in Tick, out Ready, out Go, out Cook)
2  /*@ requires true : emp @*/
3  /*@ ensures 0<=t/\t<3000 : {}.({Cook})#t.{}^* @*/
4  {
5      fork{ // The effects of the first thread is true : Ready?.{Go}
6          await Ready; emit Go;
7      }par{ // The effects of the second thread is 0<t<3000 : {Prep, Cook}#t.{Ready}
8          emit Prep;
9          async Ready { run cook (3000, Tick, Cook); }}}
10 // The final effects for main is 0<=t/\t<3000 : ({Prep}.{Cook})#t.{Ready}.{Go}
11
12 hiphop module cook (var d, in Tick, out Cook)
13 /*@ requires d>2000 : {}^*.{Prep} @*/
14 /*@ ensures 0<=t/\t<d : ({}.{Cook})#t @*/
15 { abort count(d, Tick) { yield; emit Cook; }}
```

Fig. 5. Mixed Synchronous and Asynchronous Concurrency with Function Calls.

3.2 Forward Verification

As shown in Fig. 6., we demonstrate the forward verification process of the module `main`. The program effects states are captured in the form of $\langle\Phi\rangle$. To facilitate the illustration, we label the verification steps by (1), ..., (9), and mark the deployed inference rules (cf. Sec. 5.1) in [gray].

(1) *fork*{ *(– initialize the current effects state using the module precondition –)*
$\langle true : emp\rangle$ *(– emp indicates an empty trace –)*
(2) *await* *Ready*;
$\langle true : Ready? \cdot \{\}\rangle$ [*FV-Await*]
(3) *emit* *Go*;
$\langle true : Ready? \cdot \{Go\}\rangle$ [*FV-Emit*]
(4) }*par*{ *(– initialize the current effects state using the module precondition –)*
$\langle true : emp\rangle$
(5) *emit* *Prep*;
$\langle true : \{Prep\}\rangle$ [*FV-Emit*]
(6) *async* *Ready*{
(7) *run* *cook* (*3000*, *Cook*)}}}
(-TRS: check the precondition of module cook-)
$d{=}3000 : \{Prep\} \sqsubseteq d{>}2000 \wedge \{Prep\}$
(-TRS: succeed-)
$\langle 0{\le}t{<}3000 : (\{Prep\} \cdot \{Cook\})\#t\rangle$ [*FV-Call*]
$\langle 0{\le}t{<}3000 : (\{Prep\} \cdot \{Cook\})\#t \cdot \{Ready\}\rangle$ [*FV-Async*]
(8) $\langle (true \wedge 0{\le}t{<}3000) : Ready? \cdot \{Go\} \;||\; (\{Prep\} \cdot \{Cook\})\#t \cdot \{Ready\}\rangle$ [*FV-Fork*]
$\langle 0{\le}t{<}3000 : (\{Prep\} \cdot \{Cook\})\#t \cdot \{Ready\} \cdot \{Go\}\rangle$ [*Effects-Normalization*]
(9) *(-TRS: check the postcondition of module main; Succeed, cf. Table 1.-)*
$0{\le}t{<}3000 : (\{Prep\} \cdot \{Cook\})\#t \cdot \{Ready\} \cdot \{Go\} \sqsubseteq 0{\le}t{<}3000 : \{\} \cdot (\{Cook\})\#t \cdot \{\}^\star$

Fig. 6. The forward verification example for the module `main`.

The effects states (1) and (4) are initial effects entering into the *fork*/*par* statement. The effects state (2) is obtained by [FV-Await], which concatenates a blocking signal to the current effects. The effects states (3) and (5) are obtained by [FV-Emit], which simply adds the emitted signal to the current time instance. The intermediate effects state of (7) is obtained by [FV-Call]. Before each function call, it invokes the TRS to check whether the current effects state satisfies the precondition of the callee module. If it is not satisfied, the verification fails, otherwise it concatenates the callee's postcondition to the current effects state. The final effects state of (7) is obtained by [FV-Async], which adds a new time instance $\{Ready\}$ to the current effects state, indicating that the asynchronous program has been resolved. In step (8), we parallel compose the effects from both of the branches, and normalize the final effects. After these states transformations, step (9) checks the satisfiability of the inferred effects against the declared postcondition by invoking the TRS.

3.3 The TRS

Our TRS is obligated to check the inclusions between timed synchronous effects, which is an extension of Antimirov and Mosses's algorithm. The rewriting system in [Antimirov and Mosses 1995] decides inequalities of regular expressions (REs) through an iterated process of checking the inequalities of their *partial derivatives* [Antimirov 1995]. There are two basic rules: [*DISPROVE*], which infers false from trivially inconsistent inequalities; and [*UNFOLD*], which applies Definition 3.1 to generate new inequalities.

Given Σ is the whole set of the alphabet, $D_{\underline{\mathtt{A}}}(r)$ is the partial derivative of r w.r.t the signal $\underline{\mathtt{A}}$.

Definition 3.1 (REs Inequality). For REs r and s, $r \preceq s \Leftrightarrow \forall(\underline{\mathtt{A}} \in \Sigma).\ D_{\underline{\mathtt{A}}}(r) \preceq D_{\underline{\mathtt{A}}}(s)$.

Similarly, we defined the Definition 3.2 for unfolding the inclusions between timed synchronous effects, where $D_{I\#t}(\Phi)$ is the partial derivative of Φ w.r.t the time instance I and time bound t.

Definition 3.2 (Timed Synchronous Effects Inclusion). For timed synchronous effects Φ_1 and Φ_2, $\Phi_1 \sqsubseteq \Phi_2 \Leftrightarrow \forall I.\forall t{\geq}0.\ D_{I\#t}(\Phi_1) \sqsubseteq D_{I\#t}(\Phi_2)$.

Next, we continue with the step (9) in Fig. 6., to demonstrate how the TRS handles arithmetic constraints and dependent values. As shown in Table 1., it automatically proves that the inferred effects of `main` satisfy the declared postcondition. We mark the rewriting rules (cf. Sec. 6) in [gray].

Table 1. The inclusion proving example. (For simplicity, we use 3 seconds to replace the 3000 milliseconds.)

$$
\begin{array}{rl}
t_L{<}3 \wedge t_L^1{+}t_L^2{=}t_L \wedge t_L^2{=}t_R \Rightarrow t_R{<}3 \quad emp \sqsubseteq \{\}^\star & \\
\hline
t_L{<}3 \wedge t_L^1{+}t_L^2{=}t_L \wedge t_L^2{=}t_R : emp \sqsubseteq t_R{<}3 : \bot \vee \{\}^\star & \text{⑧[PROVE]} \\
\hline
t_L{<}3 \wedge t_L^1{+}t_L^2{=}t_L \wedge t_L^2{=}t_R : \{Go\} \sqsubseteq t_R{<}3 : emp \vee \{\} \cdot \{\}^\star & \text{⑦[UNFOLD]} \\
\hline
t_L{<}3 \wedge t_L^1{+}t_L^2{=}t_L \wedge t_L^2{=}t_R : \{Go\} \sqsubseteq t_R{<}3 : \bot \vee \{\}^\star & \text{⑤[Normalization]} \\
\hline
t_L{<}3 \wedge t_L^1{+}t_L^2{=}t_L \wedge t_L^2{=}t_R : \{Ready\} \cdot \{Go\} \sqsubseteq t_R{<}3 : emp \vee \{\} \cdot \{\}^\star & \text{⑤[UNFOLD]} \\
\hline
t_L{<}3 \wedge t_L^1{+}t_L^2{=}t_L : \{Cook\}\#t_L^2 \cdot \{Ready\} \cdot \{Go\} \sqsubseteq t_R{<}3 : \{Cook\}\#t_R \cdot \{\}^\star & \text{④[UNFOLD–UNIFY]} \\
\hline
t_L{<}3 \wedge t_L^1{+}t_L^2{=}t_L : \{Prep\}\#t_L^1 \cdot \{Cook\}\#t_L^2 \cdot \{Ready\} \cdot \{Go\} \sqsubseteq t_R{<}3 : \{\} \cdot \{Cook\}\#t_R \cdot \{\}^\star & \text{③[UNFOLD]} \\
\hline
t_L{<}3 : (\{Prep\} \cdot \{Cook\})\#t_L \cdot \{Ready\} \cdot \{Go\} \sqsubseteq t_R{<}3 : \{\} \cdot \{Cook\}\#t_R \cdot \{\}^\star & \text{②[SPLIT]} \\
\hline
t{<}3 : (\{Prep\} \cdot \{Cook\})\#t \cdot \{Ready\} \cdot \{Go\} \sqsubseteq t{<}3 : \{\} \cdot \{Cook\}\#t \cdot \{\}^\star & \text{①[RENAME]}
\end{array}
$$

Note that time instance $\{Prep\}$ entails $\{\}$ because the former contains more constraints. We formally define the subsumption for time instances in Definition 6.5. Intuitively, we use [*DISPROVE*]

wherever the left-hand side (LHS) is *nullable*[7] while the right-hand side (RHS) is not. [*DISPROVE*] is the heuristic refutation step to disprove the inclusion early, improving the verification efficiency.

As shown in Table 1., in step ①, we rename the time variables to avoid the name clashes between the antecedent and the consequent. In step ②, we design a new rule [*SPLIT*] to accommodate the extended real-time constraints, which introduces two new time variables t_L^1 and t_L^2, and extends the previous constraint $t_L{<}3$ with constraints $t_L^1{+}t_L^2{=}t_L$. Then t_L^1 and t_L^2 mark the real-time constraints for time instances $\{Prep\}$ and $\{Cook\}$ respectively. Then in step ③, we eliminate $\{Prep\}$ from the LHS and $\{\}$ from the RHS, as the instances entailment $\{Prep\} \subseteq \{\}$ holds. In step ④, in order to conduct a further unfolding, we unify time variables t_L^2 and t_R by adding the constraint $t_L^2{=}t_R$. In step ⑤, from the RHS, since $\{\}^\star{=}emp \vee \{\} \cdot \{\}^\star$, eliminating one time instance $\{Ready\}$ from it will lead to $\bot \vee \{\}^\star$, which can be normalised into $\{\}^\star$ in step ⑥. At the end of the rewriting, we manage to prove that $(t_L{<}3 \wedge t_L^1{+}t_L^2{=}t_L \wedge t_L^2{=}t_R) \implies t_R{<}3$ [8]; therefore, the proof succeed.

Termination is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization* [Brotherston 2005] (cf. Table 5.).

4 LANGUAGE AND SPECIFICATIONS

4.1 The Target Language

To formulate the target language, we generalise the design of Hiphop.js into a core language λ_{HH}, which provides the infrastructure for mixing synchronous and asynchronous concurrency models. We here formally define the syntax of λ_{HH}, as shown in Fig. 7. The statements marked as purple come from the Esterel v5 [Berry 1999, 2000] endorsed by current academic compilers; while the statements marked as blue provide the asynchrony coming from the usage of JavaScript promises. In this work, we are mainly interested in signal status and control propagation, which are not related to data, therefore the data variables and data-handling primitives are abstracted away.

Meta-variables are **S**, x and nm . Basic signal types include *IN* for input signals, *OUT* for output signals, *INOUT* for both and *int* for integer variables. **var** represents the countably infinite set of arbitrary distinct identifiers. We assume that programs are well-typed conforming to basic types τ.

A program $\mathcal{P}$ comprises a list of module definitions $\overrightarrow{module}$. Here, we use the $\rightarrow$ script to denote a finite vector (possibly empty) of items. Each *module* has a name nm, a list of well-typed arguments $\overrightarrow{\tau\ x}$ and $\overrightarrow{\tau\ \mathbf{S}}$, a statement-oriented body p, associated with a precondition Φ_{pre} and a postcondition Φ_{post}. (The syntax of effects specification Φ is given in Fig. 8.)

$$
\begin{array}{llll}
(Program) & \mathcal{P} & ::= & \overrightarrow{module} \qquad (Basic\ Types)\quad \tau ::= IN \mid OUT \mid INOUT \mid int \\
(Module\ Def.) & module & ::= & nm\ (\overrightarrow{\tau\ x},\ \overrightarrow{\tau\ \mathbf{S}})\ \langle \mathbf{requires}\ \Phi_{pre}\ \mathbf{ensures}\ \Phi_{post} \rangle\ p \\
(Statement) & p,\ q & ::= & nothing \mid yield \mid emit\ \mathbf{S} \mid present\ \mathbf{S}\ p\ q \mid seq\ p\ q \mid fork\ p\ q \\
 & & & \mid loop\ p \mid run\ nm\ (\overrightarrow{x}, \overrightarrow{\mathbf{S}}) \mid abort\ p\ when\ d \\
 & & & \mid async\ \mathbf{S}\ p\ d \mid await\ \mathbf{S} \mid assert\ \Phi
\end{array}
$$

$\mathbf{S} \in$ signal variables nm, x $\in$ **var** (*Finite List*) $\rightarrow$ (*Time Bounds*) $d \in \mathbb{Z}^+$

Fig. 7. Syntax of λ_{HH}.

We here explain the intuitive semantics, while the axiomatic semantics model is defined in Sec. 5. The statement *nothing* in λ_{HH} resets all the output signals into absent. A thread of execution

[7] If the event sequence is possibly empty, i.e. contains ϵ, we call it nullable, formally defined in Definition 6.1.

[8] The proof obligations generated by the verifier are discharged using constraint solver Z3 [De Moura and Bjørner 2008].

suspends itself for the current time instance using the *yield* construct, and resumes when the next time instance started. The statement *emit* **S** broadcasts the signal **S** to be set as present and terminates instantaneously. The emission of **S** is valid for the current instance only.

The statement *present* **S** p q immediately starts p if **S** is present in the current instance; otherwise it starts q when **S** is absent. The sequence statement *seq* p q immediately starts p and behaves as p as long as p remains active. When p terminates, control is passed instantaneously to q, which determines the behaviour of the sequence from then on. *(Notice that 'emit **S1**; emit **S2**' leads to* $\{S1, S2\}$*, which emits **S1** and **S2** simultaneously and terminates instantaneously.)*

The parallel statement *fork* p q runs p and q in parallel. It remains active as long as one of its branches remains active. The parallel statement terminates when both p and q are terminated. The branches can terminate in different instances, and the parallel waits for the last one to terminate.

The statement *loop* p implements an infinite loop, but it is possible to be aborted or suspended by enclosing it within a preemptive statement. When p terminates, it is immediately restarted. The body of a loop is not allowed to terminate instantaneously when started, i.e., it must execute a yield statement to avoid an 'infinite instance'. For example, '*loop emit* **S**' is not a correct program.

The statement *run nm* $(\overrightarrow{x}, \overrightarrow{\mathbf{S}})$ is a call to module *nm*, parametrising with values and signals.

To facilitate a preemptive concurrency, as well as provide necessary real-time bounds, λ_{HH} includes the primitive statement *abort* p *when* d, which runs statement p to completion. If the execution reached the time bound d, it terminates immediately.

Async statements implement long lasting background operations. They are the essential ingredient for mixing indeterministic asynchronous computation and deterministic synchronous computations. In other words, the *async* statement enables well-behaving synchronous to regulate unsteady asynchronous computations. The statement *async* **S** p d is supposed to spawn a long lasting background computation, where d represents an execution delay. When it completes, the asynchronous block will resume the synchronous machine. Therefore when a signal **S** is specified with the async call, it emits **S** when the asynchronous block completes.

The statement *await* **S** blocks the execution and waits for the signal **S** to be emitted across the threads. The statement assert Φ is used to guarantee the temporal property Φ asserted at a certain point of the programs. [Kambona et al. 2013] shows that such a language combination makes reactive programming more powerful and flexible than the traditional web programming.

4.2 The Specification Language

We plant the effects specifications into the Hoare-style verification system, using Φ_{pre} and Φ_{post} to capture the temporal pre/post condition.

$$
\begin{array}{rrcl}
(\textit{Timed Effects}) & \Phi & ::= & \pi : es \mid \Phi_1 \vee \Phi_2 \\
(\textit{Timed Sequence}) & es & ::= & \bot \mid \epsilon \mid I \mid \mathbf{S}? \mid es_1 \cdot es_2 \mid es_1 \vee es_2 \mid es_1 || es_2 \mid es\#t \mid es^{\star} \mid es^{\omega} \\
(\textit{Time Instance}) & I & ::= & \{\} \mid \{\mathbf{S}\} \mid \{\overline{\mathbf{S}}\} \mid I_1 \cup I_2 \\
(\textit{Pure}) & \pi & ::= & \mathit{True} \mid \mathit{False} \mid A(t_1, t_2) \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg\pi \\
 & & & \mid \pi_1 \Rightarrow \pi_2 \mid \forall x.\pi \mid \exists x.\pi \\
(\textit{Real-Time Term}) & t & ::= & c \mid n \mid t_1 + t_2 \mid t_1 - t_2
\end{array}
$$

$\mathbf{S} \in$ signal variables $\quad c ::\in \mathbb{R}^+ \quad n ::\in \mathbf{var}$

(*Blocking*) ? (*Real Time Bound*) # (*Kleene Star*) $\star$ (*Infinity*) ω

Fig. 8. Syntax of Timed Synchronous Effects.

The syntax of the timed synchronous effects is formally defined in Fig. 8. Effects is a conditioned time instance sequence $\pi : es$ or a disjunction of two effects $\Phi_1 \vee \Phi_2$. Timed sequences comprise *nil* ($\bot$); an empty trace ϵ; a single time instance represented by I; a waiting for a single signal $\mathbf{S}?$; sequences concatenation $es_1 \cdot es_2$; disjunction $es_1 \vee es_2$; synchronous parallelism $es_1||es_2$.

We introduce a new operator #, and the effects $es\#t$ represents that a trace takes real-time t to complete, where t is a *term*. A time sequence can be constructed by $\star$, representing zero or more times repetition of a trace; or constructed by ω, representing an definitely infinite repetition of a trace.

There are two possible states for a signal: present $\mathbf{S}$, or absent $\overline{\mathbf{S}}$, The default state of signals in a new time instance is absent. A time instance I is a set of signals; and it can be possible empty sets $\{\}$, indicating that there is no signal constraints for the time instance.

We use π to donate a pure formula which captures the (Presburger) arithmetic conditions on terms or program parameters. We use $A(t_1, t_2)$ to represent atomic formulas of two terms (including $=$, $>$, $<$, $\geq$ and $\leq$). A term can be a constant integer value c, an integer variable n which is an input parameter of the program and can be constrained by a pure formula. A term also allows simple computations of terms, $t_1{+}t_2$ and $t_1{-}t_2$. To abstract the elapsed time, the default and implicit pure constraints of all the terms is to be greater or e qual to 0.

4.3 Semantic Model of Timed Effects

$$
\begin{array}{lll}
d, \varphi \models \Phi_1 \vee \Phi_2 & \textit{iff} & d, \varphi \models \Phi_1 \textit{ or } d, \varphi \models \Phi_2 \\
d, \varphi \models \pi : \epsilon & \textit{iff} & d{=}0 \textit{ and } \mathtt{SAT}(\pi) \textit{ and } \varphi{=}[] \\
d, \varphi \models \pi : I & \textit{iff} & d{\geq}0 \textit{ and } \mathtt{SAT}(\pi) \textit{ and } \varphi{=}[I] \\
d, \varphi \models \pi : \mathbf{S}? & \textit{iff} & d{\geq}0 \textit{ and } \mathtt{SAT}(\pi) \textit{ and } \exists n{\geq}0.\ \varphi{=}\{\overline{\mathbf{S}}\}^n{+}{+}[\{\mathbf{S}\}] \\
d, \varphi \models \pi : (es_1{\cdot}es_2) & \textit{iff} & \mathtt{SAT}(\pi) \textit{ and } \exists \varphi_1, \varphi_2, d_1, d_2 \textit{ and } \varphi{=}\varphi_1{+}{+}\varphi_2, d{=}d_1{+}d_2 \textit{ and} \\
 & & d_1, \varphi_1 \models \pi : es_1 \textit{ and } d_2, \varphi_2 \models \pi : es_2 \\
d, \varphi \models \pi : (es_1 \vee es_2) & \textit{iff} & \mathtt{SAT}(\pi) \textit{ and } d, \varphi \models \pi : es_1 \textit{ or } d, \varphi \models \pi : es_2 \\
d, \varphi \models \pi : (es_1 || es_2) & \textit{iff} & \mathtt{SAT}(\pi) \textit{ and } d, \varphi \models \pi : es_1 \textit{ and } d, \varphi \models \pi : es_2 \\
d, \varphi \models \pi : es\#t & \textit{iff} & d, \varphi \models (\pi \wedge t{=}d) : es \\
d, \varphi \models \pi : es^{\star} & \textit{iff} & d, \varphi \models \pi : \epsilon \textit{ or } d, \varphi \models \pi : (es \cdot es^{\star}) \\
d, \varphi \models \pi : es^{\omega} & \textit{iff} & d, \varphi \models \pi : (es \cdot es^{\omega}) \\
d, \varphi \models \mathit{False} : \bot & \textit{iff} & \mathit{otherwise}
\end{array}
$$

Fig. 9. Semantics of Timed Synchronous Effects.

To define the semantic model, we use φ (*a trace of sets of signals*) to represent the computation execution (or time-instance multi-trees, per se), indicating the sequential constraint of the temporal behaviour; and we use d to record the computation duration of given effects. Let $d, \varphi \models \Phi$ denote the model relation, i.e., the effects Φ take exactly d milliseconds to complete; and the linear temporal sequence φ satisfies the sequential time instances defined from Φ, with d, φ from the following concrete domains: $d \triangleq \mathbb{Z}^+$ and $\varphi \triangleq \mathit{list\ \ of\ \ I}$ (a sequence of time instances).

As shown in Fig. 9., we define the semantics of timed synchronous effects. We use [] to represent the empty sequence; ++ to represent the append operation of two traces; $[I]$ to represent the sequence only contains one time instance.

$\mathtt{SAT}(\pi)$ indicates the pure π is satisfiable, which is discharged by the constraint solver Z3 [De Moura and Bjørner 2008]. I is a list of mappings from signals to status. For example, the time instance $\{\mathbf{S}\}$ indicates that signal **S** is present regardless of the status of other non-mentioned signals, i.e., time instances which at least contain **S** to be present. Any time instance contains contradictions, such as $\{\mathbf{S}, \overline{\mathbf{S}}\}$, will lead to *false*, as a signal **S** can not be both present and absent.

5 AUTOMATED FORWARD VERIFICATION

An overview of our automated verification system is given in Fig. 10. It consists of a Hoare-style forward verifier and a TRS. The inputs of the forward verifier are HipHop.js programs annotated with temporal specifications written in timed synchronous effects (cf. Fig. 4.).

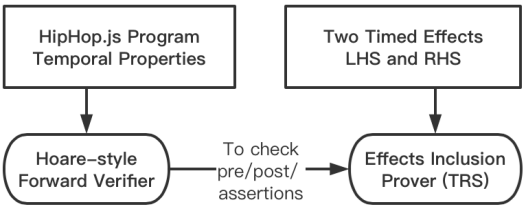

Fig. 10. System Overview.

The input of the TRS is a pair of effects LHS and RHS, referring to the inclusion LHS $\sqsubseteq$ RHS to be checked *(LHS refers to left-hand side effects, and RHS refers to right-hand side effects.)*. Besides, the verifier calls the TRS to prove produced inclusions, i.e., between the effects states and pre/post conditions or assertions (cf. Fig. 6.). The TRS will be explained in Sec. 6.

In this section, we give an axiomatic semantics model[9] for the core language λ_{HH}, by formalising a set of forward inductive rules. These rules transfer program states and systematically accumulate the effects syntactically. To define the model, we introduce an environment $\mathcal{E}$ and describe a program state in a four-elements tuple $\langle \Pi, H, C, T \rangle$, with the following concrete domains:

$$\Pi \triangleq t \rightarrow \pi, \qquad H \triangleq es, \qquad C \triangleq \overrightarrow{\mathbf{S} \rightarrow \Delta} \mid \mathbf{S}?, \qquad T \triangleq t, \qquad \mathcal{E} \triangleq \overrightarrow{\mathbf{S}}, \qquad \Delta \triangleq Present \mid Absent \mid Undef$$

Let Π represents the pure constraints for all the terms; H represents the trace of *history*; C represents the *current* time instance or a waiting signal; T is a term binding C; $\mathcal{E}$ be the environment containing all the local and output signals; Δ marks the signal status.

5.1 Forward Inference Rules

To help with reasoning logical errors (cf. Sec. 7.2.1), the rule $[FV\text{-}Nothing]$ extends the current time instance with all the signals in the environment pointing to absent.

$$\frac{C' = \{\mathbf{S} \mapsto Absent \mid \forall \mathbf{S} \in \mathcal{E}\}}{\mathcal{E} \vdash \langle \Pi, H, C, T \rangle \ nothing \ \langle \Pi, H, C{+}{+}C', T \rangle} \ [FV\text{-}Nothing]$$

The rule $[FV\text{-}Emit]$ extends the current time instance with signal **S** pointing to *Present*.

$$\frac{C' = C{+}{+}[(\mathbf{S} \mapsto Present)]}{\mathcal{E} \vdash \langle \Pi, H, C, T \rangle \ emit \ \mathbf{S} \ \langle \Pi, H, C', T \rangle} \ [FV\text{-}Emit]$$

The rule $[FV\text{-}Yield]$ archives the current time instance to the history trace; then initializes a new time instance where all the signals from $\mathcal{E}$ are set to be undefined. Then the new *current* is bound

[9] Based on the operational semantics of Esterel and JavaScript's asynchrony formally defined in [Berry and Gonthier 1992] and [Madsen et al. 2017] respectively.

with a fresh non-negative term T'.

$$\frac{\Pi'{=}\Pi \wedge T'{\geq}0 \qquad C'{=}\{\mathbf{S} \mapsto \mathit{Undef} \mid \forall \mathbf{S} \in \mathcal{E}\} \qquad H'{=}H \cdot (C\#T) \qquad (T' \ \mathit{is\ fresh})}{\mathcal{E} \vdash \langle \Pi, H, C, T\rangle \ \mathit{yield}\ \langle \Pi', H', C', T'\rangle} \ [\mathit{FV\text{-}Yield}]$$

The rule [*FV-Present*] enters into branches p and q after setting the status of $\mathbf{S}$ in the current time instance to *Present* and *Absent* respectively. We deploy the *cut* function to reallocate the next program state. Each signal can be reset from *Undef* to *Present* or *Absent* maximum once.

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle \Pi, H, C[\mathbf{S} \mapsto \mathit{Present}], T\rangle \ \ p \ \ \langle \Pi_1, H_1, C_1, T_1\rangle \\ \mathcal{E} \vdash \langle \Pi, H, C[\mathbf{S} \mapsto \mathit{Absent}], T\rangle \ \ q \ \ \langle \Pi_2, H_2, C_2, T_2\rangle \\ \langle \Pi', H', C', T'\rangle{=}\mathit{cut} \ \ (\Pi_1 \wedge \Pi_2 : H_1 \cdot (C_1\#T_1) \vee H_2 \cdot (C_2\#T_2))\end{array}}{\mathcal{E} \vdash \langle \Pi, H, C, T\rangle \ \mathit{present}\ \mathbf{S}\ p\ q\ \langle \Pi', H', C', T'\rangle} \ [\mathit{FV\text{-}Present}]$$

Definition 5.1 (Cut). Given any π and es, we define: $\mathit{cut}(\pi{:}es)$ computes $\langle \Pi, H, C, T\rangle$. In particular,

$$\mathit{cut}(\pi : es' \cdot I){=}\langle \pi \wedge T{\geq}0, es', I, T\rangle \qquad (es{=}es' \cdot I,\ T \ \mathit{is\ fresh})$$

$$\mathit{cut}(\pi : (es' \cdot I)\#t){=}\langle \pi \wedge T_1{\geq}0 \wedge T_2{\geq}0 \wedge T_1{+}T_2{=}t, es'\#T_1, I, T_2\rangle \qquad (es{=}(es' \cdot I)\#t,\ T_1,\ T_2 \ \mathit{are\ fresh})$$

The rule [*FV-Fork*] gets $\langle \Pi_1, H_1, C_1, k_1\rangle$ and $\langle \Pi_2, H_2, C_2, k_2\rangle$ by executing p and q independently. We parallel synchronise the effects from these two branches; and use the *cut* function to reallocate the next program state.

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle \Pi, H, C, T\rangle \ \ p \ \ \langle \Pi_1, H_1, C_1, T_1\rangle \qquad \mathcal{E} \vdash \langle \Pi, H, C, T\rangle \ \ q \ \ \langle \Pi_2, H_2, C_2, T_2\rangle \\ \langle \Pi', H', C', T'\rangle{=}\mathit{cut} \ \ (\Pi_1 \wedge \Pi_2 : H_1 \cdot (C_1\#T_1) \ \ ||\ \ H_2 \cdot (C_2\#T_2))\end{array}}{\mathcal{E} \vdash \langle \Pi, H, C, T\rangle \ \mathit{fork}\ p\ q\ \langle \Pi', H', C', T'\rangle} \ [\mathit{FV\text{-}Fork}]$$

The rule [*FV-Seq*] firstly gets $\langle \Pi_1, H_1, C_1, T_1\rangle$ by executing p. Then it further gets $\langle \Pi_2, H_2, C_2, T_2\rangle$ by continuously executing q, to be the final program state.

$$\frac{\mathcal{E} \vdash \langle \Pi, H, C, T\rangle \ \ p \ \ \langle \Pi_1, H_1, C_1, T_1\rangle \qquad \mathcal{E} \vdash \langle \Pi_1, H_1, C_1, T_1\rangle \ \ q \ \ \langle \Pi_2, H_2, C_2, T_2\rangle}{\varrho \vdash \langle \Pi, H, C, T\rangle \ \mathit{seq}\ p\ q\ \langle \Pi_2, H_2, C_2, T_2\rangle} \ [\mathit{FV\text{-}Seq}]$$

The rule [*FV-Loop*] computes a fixpoint (as the invariant effects of the loop body) $\langle \Pi_2, H_2, C_2, T_2\rangle$ by continuously executing p twice[10], starting from temporary initialised program states $\langle \Pi, \epsilon, C, T\rangle$ and $\langle \Pi_1, \epsilon, C_1, T_1\rangle$ respectively. The final state will contain a repeated trace $(H_2 \cdot C_2\#T_2)^\star$.

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle \Pi, \epsilon, C, T\rangle \ \ p \ \ \langle \Pi_1, H_1, C_1, T_1\rangle \qquad \mathcal{E} \vdash \langle \Pi_1, \epsilon, C_1, T_1\rangle \ \ p \ \ \langle \Pi_2, H_2, C_2, T_2\rangle \\ H'{=}H \cdot H_1 \cdot (H_2 \cdot C_2\#T_2)^\star \cdot H_2\end{array}}{\mathcal{E} \vdash \langle \Pi, H, C, T\rangle \ \mathit{loop}\ p\ \langle \Pi_2, H', C_2, T_2\rangle} \ [\mathit{FV\text{-}Loop}]$$

The rule [*FV-Call*] triggers the back-end solver TRS to check if the precondition of the callee, Φ_{pre}, is satisfied by the current effects state or not. If it holds, the rule obtains the next program state by concatenating the postcondition Φ_{post} to the current effects state. (cf. Fig. 6.)

$$\frac{\begin{array}{c}nm \ \ (\overrightarrow{\tau\, x},\ \overrightarrow{\tau\, \mathbf{S}})\ \langle \mathbf{requires}\ \Phi_{pre} \ \ \mathbf{ensures}\ \Phi_{post}\rangle\ p \in \mathcal{P} \\ \mathit{TRS} \vdash \Pi : H \cdot (C\#T) \sqsubseteq \Phi_{pre} \qquad \langle \Pi', H', C', T'\rangle{=}\mathit{cut}\ (\Pi : H \cdot (C\#T) \cdot \Phi_{post})\end{array}}{\mathcal{E} \vdash \langle \Pi, H, C, T\rangle \ \mathit{run}\ nm\ (\overrightarrow{x}, \overrightarrow{\mathbf{S}})\ \langle \Pi', H', C', T'\rangle} \ [\mathit{FV\text{-}Call}]$$

The rule [*FV-Abort*] initialises the state using $\langle \Pi, \epsilon, C, T\rangle$ before entering into p, and obtains $\langle \Pi', H', C', T'\rangle$ after the execution; then uses a fresh time variable T_d to set an upper-bound for the execution $H' \cdot (C'\#T')$, where the constraint on T_d is $0{\leq}T_d{\leq}d$. Lastly, it sets the new current

[10] The deterministic loop invariant can be fixed after the second run of the loop body, cf. Appendix C for the proof.

time instance C'' by setting all the existing signals to undefined, which is bound with a fresh non-negative term T_f.

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle \Pi, \epsilon, C, T\rangle \;\; p \;\; \langle \Pi', H', C', T'\rangle \qquad \Pi''{=}\Pi' \wedge (0{\leq}T_d{<}d) \wedge (T_f{\geq}0) \\ H''{=}(H' \cdot (C'\#T'))\#T_d \qquad C''{=}\{\mathbf{S} \mapsto \mathit{Undef} \mid \forall \mathbf{S} \in \mathcal{E}\} \qquad (T_d, T_f \;\; \mathit{are\ fresh})\end{array}}{\mathcal{E} \vdash \langle \Pi, H, C, T\rangle \; \mathit{abort}\; p \; \mathit{when}\; d \; \langle \Pi'', H \cdot H'', C'', T_f\rangle} \; [\mathit{FV\text{-}Abort}]$$

Dually, the rule [*FV-Async*] initialises the states using $\langle \Pi, \epsilon, C, k\rangle$ before entering into p, and obtains $\langle \Pi', H', C', k'\rangle$ after the execution; then uses a fresh time variable T_d to set a lower-bound for the execution $H' \cdot (C'\#T')$, where the constraint on T_d is $T_d{\geq}d$. Lastly, it creates the new current time instance C'' by setting $\mathbf{S}$ to present instantaneously, and the rest of the signals in $\mathcal{E}$ to undefined (as it emits $\mathbf{S}$ once the asynchronous execution is completed), which is bound with a fresh non-negative term T_f.

$$\frac{\begin{array}{c}\mathcal{E} \vdash \langle \Pi, \epsilon, C, T\rangle \;\; p \;\; \langle \Pi', H', C', T'\rangle \qquad (T_d, T_f \;\; \mathit{are\ fresh}) \\ \Pi''{=}\Pi' \wedge (T_d{\geq}d) \wedge (T_f{\geq}0) \qquad H''{=}(H' \cdot (C'\#T'))\#T_d \\ C''{=}\{\mathbf{S}' \mapsto \mathit{Undef} \mid \forall \mathbf{S}' \in (\mathcal{E}\backslash\mathbf{S})\} \cup \{\mathbf{S} \mapsto \mathit{Present}\}\end{array}}{\mathcal{E} \vdash \langle \Pi, H, C, T\rangle \; \mathit{async}\; \mathbf{S}\; p\; d \; \langle \Pi'', H \cdot H'', C'', T_f\rangle} \; [\mathit{FV\text{-}Async}]$$

The rule [*FV-Await*] archives the current time instance to the history trace, then sets a new current instance as a waiting for signal $\mathbf{S}$, bound with a fresh non-negative term T'.

$$\frac{\Pi'{=}\Pi \wedge T'{\geq}0 \qquad H'{=}H \cdot (C\#t) \qquad (T' \; \mathit{is\ fresh})}{\mathcal{E} \vdash \langle \Pi, H, C, T\rangle \; \mathit{await}\; \mathbf{S} \; \langle \Pi', H', \mathbf{S}?, T'\rangle} \; [\mathit{FV\text{-}Await}]$$

The rule [*FV-Assert*] simply checks if the asserted property Φ is satisfied by the current effects state. If not, a compilation error will be raised.

$$\frac{\mathit{TRS} \vdash \Pi \wedge (H \cdot (C\#T) \sqsubseteq \Phi}{\mathcal{E} \vdash \langle \Pi, H, C, T\rangle \; \mathit{assert}\; \Phi \; \langle \Pi, H, C, T\rangle} \; [\mathit{FV\text{-}Assert}]$$

6 TEMPORAL VERIFICATION VIA A TRS

The TRS is an automated entailment checker to prove language inclusions among timed synchronous effects (cf. Definition 3.2 and Table 1.). It is triggered i) prior to temporal property assertions; ii) prior to module calls for the precondition checking; and iii) at the end of verifying a module for the post condition checking. Given two effects Φ_1, Φ_2, TRS decides if the inclusion $\Phi_1 \sqsubseteq \Phi_2$ is valid.

During the effects rewriting process, the inclusions are in the form of $\Gamma \vdash \Phi_1 \sqsubseteq^{\Phi} \Phi_2$, a shorthand for: $\Gamma \vdash \Phi \cdot \Phi_1 \sqsubseteq \Phi \cdot \Phi_2$. To prove such inclusions is to check whether all the possible timed traces in the antecedent Φ_1 are legitimately allowed in the possible timed traces from the consequent Φ_2. Γ is the proof context, i.e., a set of effects inclusion hypothesis, Φ is the history effects from the antecedent that have been used to match the effects from the consequent. Note that Γ, Φ are derived during the inclusion proof. The inclusion checking is initially invoked with $\Gamma{=}\{\}$ and $\Phi{=}\mathit{True} : \epsilon$.

Effects Disjunctions. An inclusion with a disjunctive antecedent succeeds if both disjunctions entail the consequent. An inclusion with a disjunctive consequent succeeds if the antecedent entails either of the disjunctions.

$$\frac{\Gamma \vdash \Phi_1 \sqsubseteq \Phi \qquad \Gamma \vdash \Phi_2 \sqsubseteq \Phi}{\Gamma \vdash \Phi_1 \vee \Phi_2 \sqsubseteq \Phi} \; [\mathit{LHS\text{-}OR}] \qquad \frac{\Gamma \vdash \Phi \sqsubseteq \Phi_1 \quad \mathit{or} \quad \Gamma \vdash \Phi \sqsubseteq \Phi_2}{\Gamma \vdash \Phi \sqsubseteq \Phi_1 \vee \Phi_2} \; [\mathit{RHS\text{-}OR}]$$

Now, the inclusions are disjunction-free formulas. Next we provide the definitions and implementations of auxiliary functions *Nullable*(δ), *First*(*fst*) and *Derivative*(D) respectively. Intuitively, the Nullable function $\delta(es)$ returns a boolean value indicating whether *es* contains the empty

trace; the First function $fst_\pi(es)$ computes a set of possible initial time instances of $\pi : es$; and the Derivative function $D^{\pi}_{I\#t}(es)$ computes a next-state effects after eliminating one time instance I w.r.t. the execution time t from the current effects $\pi : es$.

Definition 6.1 (Nullable). Given any time sequence es, we recursively define $\delta(es)$ as:

$$\delta(es) : bool = \begin{cases} true & if\ \ \epsilon \in es \\ false & if\ \ \epsilon \notin es \end{cases}, \text{ where}$$

$$\delta(\bot)=false \qquad \delta(\epsilon)=true \qquad \delta(I)=false \qquad \delta(\mathbf{S}?)=false \qquad \delta(es^\star)=true$$
$$\delta(es_1 \cdot es_2)=\delta(es_1) \wedge \delta(es_2) \qquad \delta(es_1 \vee es_2)=\delta(es_1) \vee \delta(es_2) \qquad \delta(es^\omega)=false$$
$$\delta(es_1||es_2)=\delta(es_1) \wedge \delta(es_2) \qquad \delta(es\#t)=\delta(es)$$

To better outline our contribution, we first present the original *First* function used in Antimirov's rewriting system, denoted using fst', defined as follows:

Definition 6.2 (Antimirov's First). Let $fst'(es) := \{I \mid (I \cdot es') \in [\![es]\!]\}$ be the set of initial instances derivable from sequence es. ($[\![es]\!]$ represents all the traces contained in es.)

$$fst'(\bot)=\{\} \qquad fst'(\epsilon)=\{\} \qquad fst'(I)=\{I\} \qquad fst'(es_1 \vee es_2)=fst'(es_1) \cup fst'(es_2)$$
$$fst'(es^\star)=fst'(es) \qquad fst'(es_1 \cdot es_2)=\begin{cases} fst'(es_1) \cup fst'(es_2) & if\ \delta(es_1)=true \\ fst'(es_1) & if\ \delta(es_1)=false \end{cases}$$

As shown, fst' does not handle any arithmetic constraints, nor the real-time bounds constructor #. Next, we define the novel *First* function used in this work, denoted using fst.

Definition 6.3 (First). Let $fst_\pi(es) := \{(I, \pi, t) \mid \pi \wedge t \geq 0 : (I\#t) \cdot es' \in [\![\pi{:}es]\!]\}$ be the set of initial time instances derivable from effects $\pi{:}es$. ($[\![\pi{:}es]\!]$ represents all the time traces contained in $\pi{:}es$)

$$fst_\pi(\bot)=\{\} \qquad fst_\pi(\epsilon)=\{\} \qquad fst_\pi(I)=\{(I, \pi \wedge t \geq 0, t)\}\ (t\ is\ fresh) \qquad fst_\pi(es\#t)=fst_{pi}(es)$$
$$fst_\pi(es^\star)=fst_\pi(es) \qquad fst_\pi(es^\omega)=fst_\pi(es) \qquad fst_\pi(es_1 \vee es_2)=fst_\pi(es_1) \cup fst_\pi(es_2)$$
$$fst_\pi(\mathbf{S}?)=fst_\pi(\{\mathbf{S}\}) \cup fst_\pi(\{\overline{\mathbf{S}}\}) \qquad fst_\pi(es_1||es_2)=unify\ (fst_\pi(es_1))\ (fst_\pi(es_2))$$
$$fst_\pi(es_1 \cdot es_2)=\begin{cases} fst_\pi(es_1) \cup fst_\pi(es_2) & if\ \delta(es_1)=true \\ fst_\pi(es_1) & if\ \delta(es_1)=false \end{cases}$$

Definition 6.4 (First-Tuples Unification). Given two first tuples (I_1, π_1, t_1) and (I_2, π_2, t_2), we define that: $unify\ \ (I_1, \pi_1, t_1)\ \ (I_2, \pi_2, t_2) = (I_1 \cup I_2, \pi_1 \wedge \pi_2 \wedge (t1{=}t2), t_1)$.

Definition 6.5 (Instances Subsumption). Given two instances I and $\mathcal{J}$, we define the subset relation $I \subseteq \mathcal{J}$ as: the set of present signals in $\mathcal{J}$ is a subset of the set of present signals in I, and the set of absent signals in $\mathcal{J}$ is a subset of the set of absent signals in I[11]. Formally,

$$I \subseteq \mathcal{J} \Leftrightarrow \{\mathbf{S} \mid (\mathbf{S} \mapsto Present) \in \mathcal{J}\} \subseteq \{\mathbf{S} \mid (\mathbf{S} \mapsto Present) \in I\}$$
$$and\ \{\mathbf{S} \mid (\mathbf{S} \mapsto Absent) \in \mathcal{J}\} \subseteq \{\mathbf{S} \mid (\mathbf{S} \mapsto Absent) \in I\}$$

Definition 6.6 (Time Instances Subsumption). Given two time instances (I, π_1, t_1) and $(\mathcal{J}, \pi_2, t_2)$, we define the subset relation $(I, \pi_1, t_1) \subseteq (\mathcal{J}, \pi_2, t_2)$ as: $I \subseteq \mathcal{J}$ and $\pi_1[t_2/t_1] \Rightarrow \pi_2$.

[11] As in having more constraints refers to a smaller set of satisfying instances.

Definition 6.7 (Partial Derivative[12]*).* The partial derivative $D^{\pi}_{(I,\pi',t')}(es)$ of effects $\pi : es$ w.r.t. a time instance (I, π', t') computes the effects for the left quotient $(I, \pi', t')^{-1}[\![\pi : es]\!]$. [13]

$$D^{\pi}_{(I,\pi',t')}(\bot)=\mathit{False} : \bot \qquad D^{\pi}_{(I,\pi',t')}(\epsilon)=\mathit{False} : \bot \qquad D^{\pi}_{(I,\pi',t')}(es_1||es_2)=D^{\pi}_{(I,\pi',t')}(es_1)||D^{\pi}_{(I,\pi',t')}(es_2)$$

$$D^{\pi}_{(I,\pi',t')}(es^{\star})=D^{\pi}_{(I,\pi',t')}(es)\cdot(es^{\star}) \qquad D^{\pi}_{(I,\pi',t')}(es^{\omega})=D^{\pi}_{(I,\pi',t')}(es)\cdot(es^{\omega})$$

$$D^{\pi}_{(I,\pi',t')}(es_1\cdot es_2)=\begin{cases} D^{\pi}_{(I,\pi',t')}(es_1)\cdot es_2 \vee D^{\pi}_{(I,\pi',t')}(es_2) & \mathit{if}\ \delta(es_1)=\mathit{true} \\ D^{\pi}_{(I,\pi',t')}(es_1)\cdot es_2 & \mathit{if}\ \delta(es_1)=\mathit{false} \end{cases}$$

$$D^{\pi}_{(I,\pi',t')}(es_1 \vee es_2)=D^{\pi}_{(I,\pi',t')}(es_1) \vee D^{\pi}_{(I,\pi',t')}(es_2)$$

$$D^{\pi}_{(I,\pi',t')}(es\#t)=\pi \wedge (t_1{+}t_2{=}t) \wedge (t_1{=}t') : (D^{\pi}_{(I,\pi',t')}(es))\#t_2 \quad (\mathit{fresh}\ t_1\ t_2)$$

$$D^{\pi}_{(I,\pi',t')}(\mathbf{S}?)=\pi : \epsilon\ (I{\subseteq}\{\mathbf{S}\}) \qquad D^{\pi}_{(I,\pi',t')}(\mathbf{S}?)=\pi : \mathbf{S}?\ (I{\not\subseteq}\{\mathbf{S}\})$$

$$D^{\pi}_{(I,\pi',t')}(\mathcal{J})=\pi : \epsilon\ (I{\subseteq}\mathcal{J}) \qquad D^{\pi}_{(I,\pi',t')}(\mathcal{J})=\mathit{False} : \bot\ (I{\not\subseteq}\mathcal{J})$$

6.1 Rewriting Rules

Given the well-defined auxiliary functions above, we now discuss the key steps and related rewriting rules that we may use in such an effects inclusion proof.

(1) **Axiom rules.** Analogous to the standard propositional logic, $\bot$ (referring to *false*) entails any effects, while no *non-false* effects entails $\bot$.

$$\frac{}{\Gamma \vdash \pi : \bot \sqsubseteq \Phi}\ [\texttt{Bot-LHS}] \qquad \frac{\Phi \neq \pi : \bot}{\Gamma \vdash \Phi \not\sqsubseteq \pi : \bot}\ [\texttt{Bot-RHS}]$$

(2) **Disprove (Heuristic Refutation).** This rule is used to disprove the inclusions when the antecedent is nullable, while the consequent is not nullable. Intuitively, the antecedent contains at least one more trace (the empty trace) than the consequent. Therefore, the inclusion is invalid.

$$\frac{\delta(es_1) \wedge \neg\delta(es_2)}{\Gamma \vdash \pi_1 : es_1 \not\sqsubseteq \pi_2 : es_2}\ [\texttt{DISPROVE}] \qquad \frac{\pi_1 \Rightarrow \pi_2 \qquad \mathit{fst}_{\pi_1}(es_1) = \{\}}{\Gamma \vdash \pi_1 : es_1 \sqsubseteq \pi_2 : es_2}\ [\texttt{PROVE}]$$

(3) **Prove.** We use two rules to prove an inclusion: (i) $[PROVE]$ is used when the fst set of the antecedent is empty; and (ii) $[REOCCUR]$ to prove an inclusion when there exist inclusion hypotheses in the proof context Γ, which are able to soundly prove the current goal. One of the special cases of this rule is when the identical inclusion is shown in the proof context, we then terminate the procedure and prove it as a valid inclusion.

$$\frac{(\pi_1 : es_1 \sqsubseteq \pi_3 : es_3) \in \Gamma \qquad (\pi_3 : es_3 \sqsubseteq \pi_4 : es_4) \in \Gamma \qquad (\pi_4 : es_4 \sqsubseteq \pi_2 : es_2) \in \Gamma}{\Gamma \vdash \pi_1 : es_1 \sqsubseteq \pi_2 : es_2}\ [\texttt{REOCCUR}]$$

(4) **Unfolding (Induction).** This is the inductive step of unfolding the inclusions. Firstly, we make use of the auxiliary function *fst* to get a set of instances F, which are all the possible initial time instances from the antecedent. Secondly, we obtain a new proof context Γ' by adding the current inclusion, as an inductive hypothesis, into the current proof context

[12] Intuitively, the partial derivative refers to the left quotient of a language equation, for example, for REs, $x^{-1}[\![x \cdot y]\!]{=}y$; $y^{-1}[\![x \cdot y]\!]{=}\bot$; and $y^{-1}[\![x + y]\!]{=}\epsilon$. Here we come up with a new notion of partial derivative for timed synchronous effects.
[13] For understanding of the derivative rule for $es\#t$ (cf. Table 1. step ③), it is essentially splitting es with a head and a tail, bound with fresh terms t_1 and t_2 respectively. Meanwhile it unifies t_1 with t' and adds the constraint $t_1{+}t_2{=}t$.

Γ. Thirdly, we iterate each element $(I, \pi, t) \in F$, and compute the partial derivatives (*next-state* effects) of both the antecedent and consequent w.r.t (I, π, t). The proof of the original inclusion succeeds if all the derivative inclusions succeeds.

$$\frac{F = fst_{\pi_1}(es_1) \qquad \Gamma' = \Gamma, (\pi_1 : es_1 \sqsubseteq \pi_2 : es_2) \qquad \forall (I,\pi,t) \in F.\ (\Gamma' \vdash D^{\pi_1}_{(I,\pi,t)}(es_1) \sqsubseteq D^{\pi_2}_{(I,\pi,t)}(es_2))}{\Gamma \vdash \pi_1 : es_1 \sqsubseteq \pi_2 : es_2} \text{ [UNFOLD]}$$

(5) **Normalization.** We present a set of normalization rules to soundly transfer the timed effects into a normal form, in particular after getting their derivatives. Before getting into the above inference rules, we assume that the effects formulae are tailored accordingly based on the axioms shown in Table 2. We built the axiom system on top of a complete axiom system F_1, from (A1) to (A11), suggested by [Salomaa 1966], which was designed for regular languages. We develop axioms (A12) to (A22) to further accommodate effects constructed by $||$, # and ω.

Table 2. Normalization Axioms for Timed Synchronous Effects.

(A1)	$es_1 \vee (es_2 \vee es_3) \rightarrow (es_1 \vee es_2) \vee es_3$	(A12)	$\perp^{\omega} \rightarrow \perp$
(A2)	$es_1 \cdot (es_2 \cdot es_3) \rightarrow (es_1 \cdot es_2) \cdot es_3$	(A13)	$es^{\omega} \cdot es_1 \rightarrow es^{\omega}$
(A3)	$es_1 \vee es_2 \rightarrow es_2 \vee es_1$	(A14)	$\epsilon^{\omega} \rightarrow \perp$
(A4)	$es \cdot (es_1 \vee es_2) \rightarrow es \cdot es_1 \vee es \cdot es_2$	(A15)	$es \ \|\| \ \epsilon \rightarrow es$
(A5)	$(es_1 \vee es_2) \cdot es \rightarrow es_1 \cdot es \vee es_2 \cdot es$	(A16)	$es \ \|\| \ \perp \rightarrow \perp$
(A6)	$es \vee es \rightarrow es$	(A17)	$(I\#t) \cdot es_1 \ \|\| \ \mathcal{J} \cdot es_2 \rightarrow ((I \cup \mathcal{J})\#t) \cdot (es_1 \| es_2)$
(A7)	$es \cdot \epsilon \rightarrow es$	(A18)	$\pi : (es\#t_1)\#t_2 \rightarrow \pi \wedge (t_1{=}t_2) : es\#t_1$
(A8)	$es \cdot \perp \rightarrow \perp$	(A19)	$\epsilon\#t \rightarrow \epsilon$
(A9)	$es \vee \perp \rightarrow es$	(A20)	$\perp\#t \rightarrow \perp$
(A10)	$\epsilon \vee (es \cdot es^{\star}) \rightarrow es^{\star}$	(A21)	$False : es \rightarrow False : \perp$
(A11)	$(\epsilon \vee es)^{\star} \rightarrow es^{\star}$	(A22)	$\pi : \perp \rightarrow False : \perp$

Theorem 6.8 (Termination). *The rewriting system TRS is terminating.*

Proof. See Appendix A. □

Theorem 6.9 (Soundness). *Given an inclusion $\Phi_1 \sqsubseteq \Phi_2$, if the TRS returns TRUE when proving $\Phi_1 \sqsubseteq \Phi_2$, i.e., it has a cyclic proof, then $\Phi_1 \sqsubseteq \Phi_2$ is valid.*

Proof. See Appendix B. □

Theorem 6.10 (Completeness). *The set of axioms (A1)-(A22) is complete for timed synchronous effects.*

Proof. One can observe that by succinct applications of (A17) to (A22), we have a normal form,

$$\pi_1 : (I\#t_1) \cdot es_1 \ || \ \pi_2 : (J\#t_2) \cdot es_2 \rightarrow (\pi_1 \wedge \pi_2 \wedge t_1{=}t_2) : ((I \cup J)\#t_1) \cdot (es_1 \ || \ es_2) \tag{1}$$

Then for any ground timed synchronous effects *es*, the expression $\epsilon||es$ can be reduced to either ϵ or *es* by the applications of (A15) and (A16). The expression es^{ω} can be reduced to either $\perp$ or es^{ω} by the applications of (A12) to (A14).

Then, by the semantics defined in Fig. 9., effects constructed by Kleene star $\star$ (possibly finite and possibly infinite) are supersets of effects constructed by ω (definite infinite).

Then one can apply literally all the constructions of [Salomaa 1966] used in the completeness proof of F_1 (A1) to (A11).

□

7 IMPLEMENTATION AND CASE STUDY

7.1 Implementation

To show the feasibility of our approach, we have prototyped our automated verification system using OCaml *(A demo page and source code are available from [DemoIntro 2021]).* The proof obligations generated by the verifier are discharged using constraint solver Z3 [De Moura and Bjørner 2008]. We prove termination and correctness of the TRS. We validate the front-end forward verifier for conformance, against two implementations: the Columbia Esterel Compiler (CEC) [Stephen A. Edwards 2021] and Hiphop.js [Serrano 2021].

CEC is an open-source compiler designed for research in both hardware and software generation from the Esterel synchronous language to C or Verilog circuit description. It currently supports a subset of Esterel V5, and provides pure Esterel programs for testing. Hiphop.js's implementation facilitates the design of complex web applications by smoothly integrating Esterel and JavaScript, and provides a bench of programs for testing purposes.

Based on these two benchmarks, we validate the verifier using 155 programs, varying from 15 lines to 300 lines. We manually annotate temporal specifications in our timed effects, including both succeeded and failed instances (roughly with the a 1:1 ratio). Out of the whole test suite, 101 were from the CEC benchmark, 54 were from the Hiphop.js benchmark. Since async-await was inherited from JavaScript features, it only presents in the 54 hiphop.js programs. We conduct experiments on a MacBook Pro with a 2.6 GHz Intel Core i7 processor. Given our benchmark, the running time varying from 0 to 500 ms.

Our implementation is the first that solves the inclusion problem of symbolic timed automata, which is considered important because it overcomes the following main limitations in traditional timed model checking: i) they cannot be used to specify and verify systems incompletely specified (i. e., whose timing constants are not known yet), and hence cannot be used in early design phases; ii) verifying a system for a set of timing constants usually requires to enumerate all of them one by one if they are supposed to be integer-valued; in addition, timed automata cannot be used to verify a system for a set of timing constants that are to be taken in a real-valued dense interval.

A term rewriting system is efficient because *it only constructs automata as far as it needed*, which makes it more efficient when disproving incorrect specifications, as we can disprove it earlier without constructing the whole automata. In other words, the more incorrect specifications are, the more efficient our solver is.

7.2 Case Studies

Let's recall the existing challenges in different programming paradigms discussed in Sec. 2. In this section, we first investigate how our effects logic can help to debug errors related to both synchronous and asynchronous programs. Specifically, it effectively resolves the logical correctness checking (for synchronous languages) and critical anti-patterns checking (for premise asynchrony). Meanwhile, we further demonstrate the flexibility and expressiveness of our effects logic.

7.2.1 Logical Incorrect Catching.

We regard these programs, which have precisely one safe trace reacting to each input assignments, as logical correct. To effectively check logical correctness, in this work, given a synchronous program, after been applied to the forward rules, we compute the possible execution traces in

a disjunctive form; then prune the traces contain contradictions, following these principles: (i) explicit present and absent; (ii) each local signal should have only one status; (iii) lookahead should work for both present and absent; (iv) signal emissions are idempotent; (v) signal status should not be contradictory. Finally, upon each assignment of inputs, programs have none or multiple output traces that will be rejected, corresponding to no-valid or multiple-valid assignments.

(a)	(b)	(c)
(1) *present S1* $\langle true : \{\}\rangle$ (2) *then* $\langle true : \{S1\}\rangle$ (3) *nothing* $\langle true : \{S1, \overline{S1}\}\rangle$ (4) *else* $\langle true : \{\overline{S1}\}\rangle$ (5) *emit S1* $\langle true : \{\overline{S1}, S1\}\rangle$ (6) *end present* $\langle true : \{S1, \overline{S1}\} \vee \{\overline{S1}, S1\}\rangle$ $\langle false : \bot\rangle$	(1) *present S1* $\langle true : \{\}\rangle$ (2) *then* $\langle true : \{S1\}\rangle$ (3) *emit S1* $\langle true : \{S1, S1\}\rangle$ (4) *else* $\langle true : \{\overline{S1}\}\rangle$ (5) *nothing* $\langle true : \{\overline{S1}, \overline{S1}\}\rangle$ (6) *end present* $\langle true : \{S1, S1\} \vee \{\overline{S1}, \overline{S1}\}\rangle$ $\langle true : \{S1\} \vee \{\overline{S1}\}\rangle$	(1) *abort* $\langle true : emp\rangle$ (2) *yield* $\langle true : \{\}\rangle$ (3) *emit A* $\langle true : \{A\}\rangle$ (4) *when A* $\langle true : \{A, \overline{A}\} \vee \{\overline{A}, A\}\rangle$ $\langle false : \bot\rangle$

Table 3. Logical incorrect examples, caught by the effect logic.

As shown in Fig. 3. (a), there are no valid assignments of signal **S1** in this program. As shown in in Fig. 3. (b), it is also logical incorrect because there are two possible assignments of signal **S1** in this program. As the example shows in Fig. 3. (c), it is logical incorrect because: if the abortion does not happen, indicating **A** is absent, then the program emits **A**. If the abortion does happen, indicating **A** is present, then the program does nothing leaving **A** absent. Both way lead to contradiction.

7.2.2 *A Strange Logically Correct Program.*

Another example for synchronous languages shows that composing programs can lead to counter-intuitive phenomena. As the program shows in Fig. 11., the first parallel branch is the logical incorrect program Fig. 3. (b), while the second branch contains a non-reactive program enclosed in "present **S1**" statement. Surprisingly, this program is logical correct, since there is only one logically coherent assumption: **S1** absent and **S2** absent. With this assumption, the first present S1 statement takes its empty else branch, which justifies S1 absent. The second "present **S1**" statement also takes its empty else branch, and "emit **S2**" is not executed, which justifies **S2** absent. And our effects logic is able to soundly detect above mentioned correctness checking.

```
1  fork { present S1 then emit S1 else nothing end present
2  }par { present S1
3                 then present S2
4                        then nothing
5                        else emit S2 end present
6                 else nothing
7          end present}
```

Fig. 11. Logical Correct.

7.2.3 *Semantics of Await.*

We use Table 4. as an example to demonstrate the semantics of $A?$, i.e., "waiting for the signal A". Formally, we define,

$$A? \equiv \exists n, n{\geq}0 \wedge \{\overline{A}\}^n \cdot \{A\}$$

where$\{\overline{A}\}$ refers to all the time instances containing A to be absent.

As shown in Table 4., the LHS $\{A\} \cdot \{C\} \cdot B? \cdot \{D\}$ entails the RHS $\{A\} \cdot B? \cdot \{D\}$, as intuitively $\{C\} \cdot B?$ is a special case of $B?$. In step ①, $\{A\}$ is eliminated. In step ③, $B?$ is normalised into $\{B\} \vee (\{\overline{B}\} \cdot B?)$, By the step of ④, $\{C\}$ is eliminated together with $\{\overline{B}\}$ because $\{C\} \subseteq \{\overline{B}\}$. Now the rest part is $B? \cdot \{D\} \sqsubseteq B? \cdot \{D\}$. Here, we further normalise $B?$ from the LHS into a disjunction, leading to two proof sub-trees. From the first sub-tree, we keep unfolding the inclusion with $\{B\}$ (⑥) and $\{D\}$ (⑦) till we can prove it. Continue with the second sub-tree, we unfold it with $\{\overline{B}\}$; then in step ⑧ we observe the proposition is isomorphic with one of the the previous step, marked using (‡). We prove it using the [*REOCCUR*] rule and finish the whole writing process.

Table 4. The example for Await.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon \sqsubseteq \epsilon}\,⑦[PROVE]}{\{D\} \sqsubseteq \{D\}}}{\{B\} \cdot \{D\} \sqsubseteq (\{B\} \vee \bot) \cdot \{D\}}\,⑥[UNFOLD] \quad \dfrac{}{B? \cdot \{D\} \sqsubseteq B? \cdot \{D\}}\,(‡)}{\{\overline{B}\} \cdot B? \cdot \{D\} \sqsubseteq (\bot \vee (\{\overline{B}\} \cdot B?)) \cdot \{D\}}\,⑧[REOCCUR]}{B? \cdot \{D\} \sqsubseteq B? \cdot \{D\}}\,⑤[Disj\text{-}L] \quad (‡)}{\{C\} \cdot B? \cdot \{D\} \sqsubseteq (\bot \vee (\{\overline{B}\} \cdot B?)) \cdot \{D\}}\,④[Normalisation]}{\{C\} \cdot B? \cdot \{D\} \sqsubseteq (\{B\} \vee (\{\overline{B}\} \cdot B?)) \cdot \{D\}}\,③[UNFOLD]}{\{A\} \cdot \{C\} \cdot B? \cdot \{D\} \sqsubseteq \{A\} \cdot B? \cdot \{D\}}\,②[Normalisation]}{True : \{A\} \cdot \{C\} \cdot B? \cdot \{D\} \sqsubseteq True : \{A\} \cdot B? \cdot \{D\}}\,①[UNFOLD]$$

7.2.4 *Broken Promises Chain.*

As the prior work [Alimadadi et al. 2018; Madsen et al. 2017] present, one of the critical issues of using promise is the broken chain of the interdependent promises; and they propose the *promise graph*, as a graphical aid, to understand and debug promise-based code.

Definition 7.1 (Well-Synchronised Traces). Based on the syntax (Sec. 4.3) and semantics (Sec. 4.2) defined for timed synchronous effects, in this paper, we call traces without any blocking signals well-synchronised traces.

We here show that our algebraic effects can capture non well-synchronised traces during the rewriting process by computing the derivative. For example, the parallel composition of traces:

$$\{\mathbf{A}\} \cdot \{\mathbf{B}\} \cdot \{\mathbf{C}\} \cdot \{\mathbf{D}\} \;||\; \{\mathbf{E}\} \cdot \;\mathbf{C}? \cdot \{\mathbf{F}\}$$

leads to the final behaviour of $\{\mathbf{A}, \mathbf{E}\} \cdot \{\mathbf{B}\} \cdot \{\mathbf{C}\} \cdot \{\mathbf{D}, \mathbf{F}\}$, which is well-synchronised for all the time instances. However, if we were composing traces:

$$\{\mathbf{A}\} \cdot \{\mathbf{B}\} \cdot \{\mathbf{D}\} \;||\; \{\mathbf{E}\} \cdot \;\mathbf{C}? \cdot \{\mathbf{F}\}$$

due to the reasons that forgetting to emit **C** (In JavaScript, it could be the case that forgetting to explicitly return a promise result), it leads to a problematic trace $\{\mathbf{A}, \mathbf{E}\} \cdot \{\mathbf{B}\} \cdot \{\mathbf{D}\} \cdot \mathbf{C}? \cdot \{\mathbf{F}\}$. The final effects contain a dangling signal waiting of **C**, which indicates the corresponding anti-pattern.

7.2.5 *Handling Finite and Infinite Effects Simultaneously.*

To further demonstrate the expressiveness of our timed effects, we use an example shows in Fig. 12. It declares a recursive module named *send*, making a deterministic choice depending on input *n*: in one case it emits one signal **Done**; otherwise it emits a signal **Send**, then makes a recursive call with parameter $n-1$. Note that the post condition contains both finite effects and infinite effects in one single formula, separated by arithmetic constrains.

We here bring up the discussion distinguishing our work from the closely related prior work, which deploys an effects tuple (Φ_u, Φ_v) [Nanjo et al. 2018], separating the finite (Φ_u) and infinite (Φ_v) effects, therefore it leads to separated sets of effects semantics, and a separated reasoning on inductive and co-inductive definitions. In our work, by merging finite and infinite effects into a single disjunctive form, it not only eliminates duplicate operators, but also enables an effective reasoning of both effects logics simultaneously. The inclusion checking process on the *send*'s postcondition is demonstrated in Table 5. [REOCCUR], which finds the syntactic identity, as a companion, of the current open goal, as a bud, from the internal proof tree [Brotherston 2005].

```
1  hiphop module send (var n, out Send, out Done)
2  /*@ requires true : emp @*/
3  /*@ ensures (n>=0 : {}^*. {Done})
4           \/ (n<0 : {Send}^w) @*/
5  {
6      if (n==0) { emit Done; }
7      else {
8          emit Send;
9          yield;
10         send (d, n-1, Send, Done); }}
```

Fig. 12. A recursive `Send module`.

Table 5. Term Rewriting for Fig. 12. (We use (†) to indicate the pairing of buds with companions.)

$$\dfrac{\dfrac{\dfrac{n{=}0 \Rightarrow n{\geq}0 \qquad emp \subseteq \{\}^\star \quad [PROVE]}{n{=}0 : emp \sqsubseteq n{\geq}0 : \bot \vee \{\}^\star \vee emp}}{n{=}0 : \{Done\} \sqsubseteq n{\geq}0 : \{\}^\star \cdot \{Done\}}}{n{=}0 : \{Done\} \sqsubseteq n{\geq}0 : \{\}^\star \cdot \{Done\} \vee n{<}0 : \{Send\}^\omega} \qquad \dfrac{\dfrac{\dfrac{n{\neq}0 : \{Send\}^\omega \sqsubseteq n{<}0 : \{Send\}^\omega (\dagger)}{n{\neq}0 : \{Send\}^\omega \sqsubseteq n{<}0 : \{Send\}^\omega (\dagger)} [REOCCUR]}{n{\neq}0 : \{Send\} \cdot \{Send\}^\omega \sqsubseteq n{<}0 : \{Send\}^\omega}}{n{\neq}0 : \{Send\} \cdot \Phi^{send}_{post} \sqsubseteq n{<}0 : \{Send\}^\omega}$$

$$\dfrac{n{=}0 : \{Done\} \sqsubseteq n{\geq}0 : \{\}^\star \cdot \{Done\} \vee n{<}0 : \{Send\}^\omega \qquad n{\neq}0 : \{Send\} \cdot \Phi^{send}_{post} \sqsubseteq n{\geq}0 : \{\}^\star \cdot \{Done\} \vee n{<}0 : \{Send\}^\omega}{n{=}0 : \{Done\} \vee n{\neq}0 : \{Send\} \cdot \Phi^{send}_{post} \sqsubseteq n{\geq}0 : \{\}^\star \cdot \{Done\} \vee n{<}0 : \{Send\}^\omega}$$

7.2.6 *Prove it when Reoccur.*

Termination of TRS is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization* [Brotherston 2005], i.e., the rule [*REOCCUR*].

As shown in Table 6., the LHS formula is a disjunction of two effects. An inclusion with a disjunctive antecedent succeeds if both disjunctions entail the consequent, shown in step ①. In step ② of the first sub-tree, in order to eliminate the first instance $\{B\}$, $\{A\}^\star \# t_R$ has to be reduced to be ϵ, therefore the time constraints have nothing to do in this branch. Then in steps ④, we can simply prove it with the side constraint entailment $True \Rightarrow True$ succeed.

Looking at the second sub-tree, in step ⑤, t_L and t_R are firstly split into $t^1_L{+}t^2_L$ and $t^1_R{+}t^2_R$. Then in step ⑥, $\{A\}\#t^1_L$ together with $\{A\}\#t^1_R$ are eliminated, unifying t^1_L and t^1_R (adding the constraint $t^1_L{=}t^1_R$). In step ⑦, we observe the proposition is isomorphic with one of the the previous step, marked using (‡). Hence we apply the rule [*REOCCUR*] to prove it with a succeed side constraints entailment: $t^1_L{+}t^2_L{=}t_L \wedge t_L{<}3 \wedge t_R{=}t^1_R{+}t^2_R \wedge t^1_L{=}t^1_R \wedge t^2_L{=}t^2_R \Rightarrow t_R{<}4$.

Table 6. The reoccurrence proving example. I : The main rewriting proof tree; II : Right hand side sub-tree of the rewriting process.

$$
I:\ \dfrac{\dfrac{\dfrac{\dfrac{True \Rightarrow True \qquad \epsilon \sqsubseteq \epsilon}{True : \epsilon \sqsubseteq True : \epsilon}\ ④[PROVE]}{True : \cancel{\{B\}} \sqsubseteq True : \cancel{\epsilon \cdot \{B\}}}\ ③[Normalisation]}{True : \{B\} \sqsubseteq t_R{<}4 : \{A\}^{\star}\#t_R \cdot \{B\}}\ ②[UNFOLD] \qquad \dfrac{}{t_L{<}3 : \{A\}^{\star}\#t_L \cdot \{B\} \sqsubseteq t_R{<}4 : \{A\}^{\star}\#t_R \cdot \{B\}}\ II}{t_L{<}3 : \{A\}^{\star}\#t_L \cdot \{B\} \vee True : \{B\} \sqsubseteq t_R{<}4 : \{A\}^{\star}\#t_R \cdot \{B\}}\ ①[Disj\text{-}L]
$$

$$
II:\ \dfrac{\dfrac{\dfrac{\dfrac{t_L^1{+}t_L^2{=}t_L \wedge t_L{<}3 \wedge t_R{=}t_R^1{+}t_R^2 \wedge t_L^1{=}t_R^1 \wedge t_L^2{=}t_R^2 \Rightarrow t_R{<}4}{\cdots \wedge \cdots : \{A\}^{\star}\#t_L^2 \cdot \{B\} \sqsubseteq t_R{<}4 : \{A\}^{\star}\#t_R^2 \cdot \{B\}}\ (\ddagger)}{t_L^1{=}t_R^1 : \cancel{\{A\}\#t_L^1} \cdot \{A\}^{\star}\#t_L^2 \cdot \{B\} \sqsubseteq t_R{<}4 : \cancel{\{A\}\#t_R^1} \cdot \{A\}^{\star}\#t_R^2 \cdot \{B\}}\ ⑦[REOCCUR]}{t_L^1{+}t_L^2{=}t_L \wedge t_R{=}t_R^1{+}t_R^2 : (\{A\}\#t_L^1 \cdot \{A\}^{\star}\#t_L^2) \cdot \{B\} \sqsubseteq t_R{<}4 : (\{A\}\#t_R^1 \cdot \{A\}^{\star}\#t_R^2) \cdot \{B\}}\ ⑥[UNFOLD]}{t_L{<}3 : \{A\}^{\star}\#t_L \cdot \{B\} \sqsubseteq t_R{<}4 : \{A\}^{\star}\#t_R \cdot \{B\}\ (\ddagger)}\ ⑤[SPLIT]
$$

7.3 Discussion

As the examples show, our proposed effects logic and the abstract semantics for λ_{HH} not only tightly capture the behaviours of a mixed synchronous and asynchronous concurrency model but also help to mitigate the programming challenges in each paradigm. Meanwhile, the inferred temporal traces from a given reactive program enable a compositional timed temporal verification at the source level, which is not yet supported by existing temporal verification techniques.

8 RELATED WORK

This work is related to i) semantics of synchronous languages and asynchronous promises; ii) research on real-time system modelling and verification; and iii) existing traces-based effects systems.

8.1 Semantics of Esterel and JavaScript's asynchrony

The web orchestration language HipHop.js [Berry and Serrano 2020] integrates Esterel's synchrony with JavaScript's asynchrony, which provides the infrastructure for our work on mixed synchronous and asynchronous concurrency models. To the best of authors' knowledge, the inference rules in this work formally define the first axiomatic semantics for a core language of HipHop.js, which are established on top of the existing semantics of Esterel and JavaScript's asynchrony.

For the pure Esterel, the communication kernel of the Esterel synchronous reactive language, prior work gave two semantics, a macrostep logical semantics called the behavioural semantics [Berry 1999], and a small-step semantics called execution/operational semantics [Berry and Gonthier 1992]. Our timed synchronous effects of Esterel primitives closely follow the work of states-based semantics [Berry 1999]. In particular, we borrow the idea of internalizing state into effects using *history* instance trace and *current* time instance, that bind a partial store embedded at any level in a program. However, as the existing semantics are not ideal for compositional reasoning in terms of the source program, our forward verifier can help meet this requirement for better modularity.

In JavaScript programs, the primitives *async* and *await* serve for *promises*-based (supported in ECMAScript 6 [Ecma 1999]) asynchronous programs, which can be written in a synchronous style, leading to more scalable code. However, the ECMAScript 6 standard specifies the semantics of promises informally and in operational terms, which is not a suitable basis for formal reasoning or

program analysis. Prior work [Alimadadi et al. 2018; Madsen et al. 2017], in order to understand promise-related bugs, present the λ_p calculus, which provides a formal semantics for JavaScript promises. Based on these, our work defines the semantics of *async* and *await* in the event-driven synchronous concurrent context. In particular, the asynchronous primitive *setTimeout* provides the lower real-time bounds, cooperating with the preemptive primitive *abort* from Esterel, which provides the upper real-time bounds for the program execution.

8.2 Specifications and Real-Time Verification

Compositional specification for real-time systems based on timed process algebras has been extensively studied. Examples include the CCS+Time [Yi 1991] and Timed CSP [Dong et al. 2008]. The differences are: i) in CCS+Time, if there is no timed constraints specified, transitions takes no time; whereas in our effects logic, it means *true* (arbitrary time), which is more close to the real life context where the time bounds are often unpredictable. ii) Timed CSP does not allow explicit representation of real-time through the manipulation of clock variables.

There have been a number of translation-based approaches on building verification support for timed process algebras. For example, Timed CSP [Dong et al. 2008] is translated to Timed Automata (TA) so that the model checker Uppaal can be [Larsen et al. 1997] applied. TA are finite state automata equipped with clock variables. Models based on TA often have simple structures. For example, the input models of the Uppaal are networks of TA with no hierarchy.

In practice, system requirements are often structured into phases, which are then composed in many different ways, while TA is deficiency in modelling complex compositional systems. Users often need to manually cast high-level requirements into a set of clock variables with carefully calculated clock constraints, which is tedious and error-prone [Sun et al. 2013]. On the other hand, all the translation-based approaches share the common problem: the overhead introduced by the complex translation makes it particularly inefficient when *disproving* properties.

We believe in that the goal of verifying real-time systems, in particular safety-critical systems is to check logical temporal properties, which can be done without constructing the whole reachability graph or the full power of model-checking. We are of the opinion that our approach is simpler as it is based directly on constraint-solving techniques and can be fairly efficient in verifying systems consisting of many components as it avoids to explore the whole state-space [Song and Chin 2020; Yi et al. 1995].

Moreover, our timed synchronous effects also draws similarities to Synchronous Kleene Algebra (SKA) [Prisacariu 2010]. Kleene algebra (KA) is a decades-old sound and complete equational theory of regular expressions. Among many extensions of KA, SKA is KA extended with a synchrony combinator for actions. In particularly, SKA's *demanding relation* can be reflected by our *instances subsumption* (Definition 6.5), which contributes to the inductive unfolding process. Yet our timed effects further capture the time bounds constraints by adding the operator #, adapting to real-time system modelling and verification, which is not covered by any existing KA variants/extensions.

While the original equivalence checking algorithm for SKA terms in [Prisacariu 2010] has relied on well-studied decision procedures based on classical Thompson ϵ-NFA construction, prior work [Broda et al. 2015] shows that the use of Antimirov's partial derivatives could result in better average-case algorithms for automata construction. Moreover, between TRS and the construction of efficient automata, prior work has recently shown in [Song and Chin 2020] that the former has a minor performance advantage (over a benchmark suite) when it is compared with state-of-the-art PAT [Sun et al. 2009] model checker. Improvement came from the avoidance of the more expensive automata construction process.

8.3 Existing Traces-Based Effects Systems

Combining program events with a temporal program logic for asserting properties of event traces yields a powerful and general engine for enforcing program properties. Results in [Marriott et al. 2003; Skalka and Smith 2004; Skalka et al. 2008] have demonstrated that static approximations of program event traces can be generated by type and effect analyses [Amtoft et al. 1999; Talpin and Jouvelot 1994], in a form amenable to existing model-checking techniques for verification. We call these approximations trace-based effects.

Trace-based analyses have been shown capable of statically enforcing flow-sensitive security properties such as safe locking behaviour [Foster et al. 2002] and resource usage policies such as file usage protocols and memory management [Marriott et al. 2003]. In [Bartoletti et al. 2005], a trace effect analysis is used to enforce secure service composition. Stack-based security policies are also amenable to this form of analysis, as shown in [Skalka and Smith 2004].

More related to our work, prior research has been extending Hoare logic with event traces. The work [Malecha et al. 2011] focuses on finite traces (terminating runs) for web applications, leaving the divergent computation, which indicates *false*, verified for every specification. The work [Nakata and Uustalu 2010] focuses on infinite traces (non-terminating runs) by providing coinductively trace definitions. More recent works [Bubel et al. 2015; Song and Chin 2020] proposed dynamic logics and unified operators to reason about possibly finite and infinite traces at the same time.

Moreover, this paper draws similarities to *contextual effects* [Neamtiu et al. 2008], which computes the effects that have already occurred as the prior effects. The effects of the computation yet to take place as the future effects. Besides, prior work [Nielson et al. 1998] proposes an annotated type and effect system and infers behaviours from CML [Reppy 1993] programs for channel-based communications, though it did not provide any inclusion solving process.

However, none of the previous works have considered adding real-time abstractions into the modelling for a more extensive timed verification, which is our work's main contribution.

9 CONCLUSION

We define the syntax and semantics of the novel timed synchronous effects, to capture reactive program behaviours and temporal properties. We demonstrate how to give a rather axiomatic semantics to λ_{HH} by timed-trace processing functions. We use this semantic model to enable a Hoare-style forward verifier, which computes the program effects constructively. We present an effects inclusion checker (the TRS) to prove the annotated temporal properties efficiently. We prototype the verification system and show its feasibility. To the best of our knowledge, our work is the first that formulates semantics of a mixed Sync-Async concurrency paradigm; and that automates modular timed verification for reactive programs using an expressive effects logic.

REFERENCES

2021. https://en.wikipedia.org/wiki/Synchronous_programming_language.

Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. 2018. Finding broken promises in asynchronous JavaScript programs. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–26.

Marco Almeida, Nelma Moreira, and Rogério Reis. 2009. Antimirov and Mosses's rewrite system revisited. *International Journal of Foundations of Computer Science* 20, 04 (2009), 669–684.

Torben Amtoft, Hanne Riis Nielson, and Flemming Nielson. 1999. *Type and effect systems: behaviours for concurrency.* World Scientific.

Valentin Antimirov. 1995. Partial derivatives of regular expressions and finite automata constructions. In *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 455–466.

Valentin M Antimirov and Peter D Mosses. 1995. Rewriting extended regular expressions. *Theoretical Computer Science* 143, 1 (1995), 51–72.

Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. 2005. Enforcing secure service composition. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. IEEE, 211–223.

Gérard Berry. 1993. Preemption in concurrent systems. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 72–93.

Gerard Berry. 1999. The constructive semantics of pure Esterel-draft version 3. *Draft Version* 3 (1999).

Gérard Berry. 2000. *The Esterel v5 language primer: version v5_91*. Centre de mathématiques appliquées, Ecole des mines and INRIA.

Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming* 19, 2 (1992), 87–152.

Gérard Berry and Manuel Serrano. 2020. HipHop. js:(A) Synchronous reactive web programming.. In *PLDI*. 533–545.

Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. 2012. Pause n Play: Formalizing Asynchronous C sharp. In *European Conference on Object-Oriented Programming*. Springer, 233–257.

Sabine Broda, Sílvia Cavadas, Miguel Ferreira, and Nelma Moreira. 2015. Deciding synchronous Kleene algebra with derivatives. In *International Conference on Implementation and Application of Automata*. Springer, 49–62.

James Brotherston. 2005. Cyclic proofs for first-order logic with inductive definitions. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 78–92.

Richard Bubel, Crystal Chang Din, Reiner Hähnle, and Keiko Nakata. 2015. A dynamic logic with traces and coinduction. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 307–322.

Andrei Damian, Cezara Drăgoi, Alexandru Militaru, and Josef Widder. 2019. Communication-closed asynchronous protocols. In *International Conference on Computer Aided Verification*. Springer, 344–363.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Demo1. 2021. http://loris-5.d2.comp.nus.edu.sg/MixedSyncAsync/index.html?ex=paper_example&type=hh&options=sess.

Demo2. 2021. http://loris-5.d2.comp.nus.edu.sg/MixedSyncAsync/verify.html?ex=paper_example1.826.hh&type=hh&options=sess.

DemoIntro. 2021. http://loris-5.d2.comp.nus.edu.sg/MixedSyncAsync/introduction.html.

Jin Song Dong, Ping Hao, Shengchao Qin, Jun Sun, and Wang Yi. 2008. Timed automata patterns. *IEEE Transactions on Software Engineering* 34, 6 (2008), 844–859.

ECMA Ecma. 1999. 262: Ecmascript language specification. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr*, (1999).

Jeffrey S Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 1–12.

KLAUS V Gleissenthall, RAMI GÖKHAN Kici, ALEXANDER Bakst, DEIAN Stefan, and RANJIT Jhala. 2019. Pretend synchrony. POPL.

Dag Hovland. 2012. The inclusion problem for regular expressions. *J. Comput. System Sci.* 78, 6 (2012), 1795–1813.

Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. 2013. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. 1–9.

Matthias Keil and Peter Thiemann. 2014. Symbolic solving of extended regular expression inequalities. *arXiv preprint arXiv:1410.3227* (2014).

Kim G Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a nutshell. *International journal on software tools for technology transfer* 1, 1-2 (1997), 134–152.

Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A model for reasoning about JavaScript promises. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–24.

Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. 2011. Trace-based verification of imperative programs with I/O. *Journal of Symbolic Computation* 46, 2 (2011), 95–118.

Kim Marriott, Peter J Stuckey, and Martin Sulzmann. 2003. Resource usage verification. In *Asian Symposium on Programming Languages and Systems*. Springer, 212–229.

Keiko Nakata and Tarmo Uustalu. 2010. A Hoare logic for the coinductive trace-based big-step semantics of While. In *European Symposium on Programming*. Springer, 488–506.

Yoji Nanjo, Hiroshi Unno, Eric Koskinen, and Tachio Terauchi. 2018. A fixpoint logic and dependent effects for temporal property verification. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 759–768.

Iulian Neamtiu, Michael Hicks, Jeffrey S Foster, and Polyvios Pratikakis. 2008. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 37–49.

Hanne Riis Nielson, Torben Amtoft, and Flemming Nielson. 1998. Behaviour analysis and safety conditions: a case study in CML. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 255–269.

Cristian Prisacariu. 2010. Synchronous kleene algebra. *The Journal of Logic and Algebraic Programming* 79, 7 (2010), 608–635.
John H Reppy. 1993. Concurrent ML: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*. Springer, 165–198.
Arto Salomaa. 1966. Two complete axiom systems for the algebra of regular events. *Journal of the ACM (JACM)* 13, 1 (1966), 158–169.
Manuel Serrano. 2021. https://github.com/manuel-serrano/hiphop.
Christian Skalka and Scott Smith. 2004. History effects and verification. In *Asian Symposium on Programming Languages and Systems*. Springer, 107–128.
Christian Skalka, Scott Smith, and David Van Horn. 2008. Types and trace effects of higher order programs. *Journal of Functional Programming* 18, 2 (2008), 179–249.
Yahui Song and Wei-Ngan Chin. 2020. Automated temporal verification of integrated dependent effects. In *International Conference on Formal Engineering Methods*. Springer, 73–90.
Yahui Song and Wei-Ngan Chin. 2021. A Synchronous Effects Logic for Temporal Verification of Pure Esterel. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 417–440.
Jia Zeng Stephen A. Edwards, Cristian Soviani. 2021. http://www.cs.columbia.edu/~sedwards/cec/.
Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. 2013. Modeling and verifying hierarchical real-time systems using stateful timed CSP. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 1 (2013), 1–29.
Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. 2009. PAT: Towards flexible verification under fairness. In *International conference on computer aided verification*. Springer, 709–714.
Jean-Pierre Talpin and Pierre Jouvelot. 1994. The type and effect discipline. *Information and computation* 111, 2 (1994), 245–296.
Ghaith Tarawneh and Andrey Mokhov. 2018. Formal Verification of Mixed Synchronous Asynchronous Systems using Industrial Tools. In *2018 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. IEEE, 43–50.
Colin Vidal, Gérard Berry, and Manuel Serrano. 2018. Hiphop. js: a language to orchestrate web applications. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2193–2195.
Reinhard Von Hanxleden, Timothy Bourke, and Alain Girault. 2017. Real-time ticks for synchronous programming. In *2017 Forum on Specification and Design Languages (FDL)*. IEEE, 1–8.
Wang Yi. 1991. CCS+ time= an interleaving model for real time systems. In *International Colloquium on Automata, Languages, and Programming*. Springer, 217–228.
Wang Yi, Paul Pettersson, and Mats Daniels. 1995. Automatic verification of real-time communicating systems by constraint-solving. In *Formal Description Techniques VII*. Springer, 243–258.

A TERMINATION PROOF

Proof. Let Set$[\mathcal{I}]$ be a data structure representing the sets of inclusions.

We use S to denote the inclusions to be proved, and H to accumulate "inductive hypotheses", i.e., $S, H \in Set[\mathcal{I}]$.

Consider the following partial ordering $\succ$ on pairs $\langle S, H\rangle$:

$$\langle S_1, H_1\rangle \succ \langle S_2, H_2\rangle \text{ iff } |H_1| < |H_2| \vee (|H_1| = |H_2| \wedge |S_1| > |S_2|).$$

where $|X|$ stands for the cardinality of a set X. Let $\Rightarrow$ donate the rewrite relation, then $\Rightarrow^*$ denotes its reflexive transitive closure. For any given S_0, H_0, this ordering is well founded on the set of pairs $\{\langle S, H\rangle | \langle S_0, H_0\rangle \Rightarrow^* \langle S, H\rangle\}$, due to the fact that H is a subset of the finite set of pairs of all possible derivatives in initial inclusion.

Inference rules in our TRS given in Sec. 6.1 transform current pairs $\langle S, H\rangle$ to new pairs $\langle S', H'\rangle$. And each rule either increases $|H|$ (Unfolding) or, otherwise, reduces $|S|$ (Axiom, Disprove, Prove), therefore the system is terminating.

□

B SOUNDNESS PROOF

Proof. For each inference rules, if inclusions in their premises are valid, and their side conditions are satisfied, then goal inclusions in their conclusions are valid.

(1) **Axiom Rules:**

$$\frac{}{\Gamma \vdash \pi : \bot \sqsubseteq \Phi}\ [\texttt{Bot-LHS}] \qquad \frac{\Phi \neq \pi : \bot}{\Gamma \vdash \Phi \not\sqsubseteq \pi : \bot}\ [\texttt{Bot-RHS}]$$

- It is easy to verify that antecedent of goal entailments in the rule [*Bot-LHS*] is unsatisfiable. Therefore, these entailments are evidently valid.
- It is easy to verify that consequent of goal entailments in the rule [*Bot-RHS*] is unsatisfiable. Therefore, these entailments are evidently invalid.

(2) **Disprove Rules:**

$$\frac{\delta(es_1) \wedge \neg\delta(es_2)}{\Gamma \vdash \pi_1 : es_1 \not\sqsubseteq \pi_2 : es_2}\ [\texttt{DISPROVE}] \qquad \frac{\pi_1 \Rightarrow \pi_2 \qquad fst_{\pi_1}(es_1) = \{\}}{\Gamma \vdash \pi_1 : es_1 \sqsubseteq \pi_2 : es_2}\ [\texttt{PROVE}]$$

- It's straightforward to prove soundness of the rule [*DISPROVE*], Given that $es1$ is nullable, while es_2 is not nullable, thus clearly the antecedent contains more event traces than the consequent. Therefore, these entailments are evidently invalid.

(3) **Prove Rules:**

$$\frac{(\pi_1 : es_1 \sqsubseteq \pi_3 : es_3) \in \Gamma \qquad (\pi_3 : es_3 \sqsubseteq \pi_4 : es_4) \in \Gamma \qquad (\pi_4 : es_4 \sqsubseteq \pi_2 : es_2) \in \Gamma}{\Gamma \vdash \pi_1 : es_1 \sqsubseteq \pi_2 : es_2}\ [\texttt{REOCCUR}]$$

- To prove soundness of the rule [*PROVE*], we consider an arbitrary model, d, φ such that: $d, \varphi \models \pi_1 : es_1$. Given the side conditions from the promises, we get $d, \varphi \models \pi_2 : es_1$. When the *fst* set of es_1 is empty, es_1 is possible $\bot$ or ϵ; and es_2 is nullable. For both cases, the inclusion is valid.
- To prove soundness of the rule [*REOCCUR*], we consider an arbitrary model, d, φ such that: $d, \varphi \models \pi_1 : es_1$. Given the promises that $\pi_1 : es_1 \sqsubseteq \pi_3 : es_3$, we get $d, \varphi \models \pi_3 : es_3$; Given the promise that there exists a hypothesis $\pi_3 : es_3 \sqsubseteq \pi_4 : es_4$, we get $d, \varphi \models \pi_4 : es_4$; Given the promises that $\pi_4 : es_4 \sqsubseteq \pi_2 : es_2$, we get $d, \varphi \models \pi_2 : es_2$. Therefore, the inclusion is valid.

(4) **Unfolding Rule:**

$$\frac{\begin{array}{c} F = fst_{\pi_1}(es_1) \qquad \Gamma' = \Gamma, (\pi_1 : es_1 \sqsubseteq \pi_2 : es_2) \\ \forall (I, \pi, t) \in F.\ (\Gamma' \vdash\ D^{\pi_1}_{(I,\pi,t)}(es_1) \sqsubseteq D^{\pi_2}_{(I,\pi,t)}(es_2)) \end{array}}{\Gamma \vdash \pi_1 : es_1 \sqsubseteq \pi_2 : es_2}\ [\texttt{UNFOLD}]$$

- To prove soundness of the rule [*UNFOLD*], we consider an arbitrary model, d_1, φ_1 and d_2, φ_2 such that: $d_1, \varphi_1 \models \pi_1 : es_1$ and $d_2, \varphi_2 \models \pi_2 : es_2$. For an arbitrary time instance (I, π, t), let $d_1', \varphi_1' \models (\mathtt{I}, \pi, \mathtt{t})^{-1}[\![\pi_1 : es_1]\!]$; and $d_2', \varphi_2' \models (\mathtt{I}, \pi, \mathtt{t})^{-1}[\![\pi_2 : es_2]\!]$.
Case 1), $(\mathtt{I}, \pi, \mathtt{t}) \notin F$, $d_1', \varphi_1' \models \bot$, thus automatically $d_1', \varphi_1' \models D^{\pi_2}_{(\mathtt{I},\pi,\mathtt{t})}(es_2)$;
Case 2), $(\mathtt{I}, \pi, \mathtt{t}) \in F$, given that inclusions in the rule's premise is valid, then $d_1', \varphi_1' \models D^{\pi_2}_{(\mathtt{I},\pi,\mathtt{t})}(es_2)$.
By Definition 3.2, since for all (I, π, t), $D^{\pi_1}_{(\mathtt{I},\pi,\mathtt{t})}(es_1) \sqsubseteq D^{\pi_2}_{(\mathtt{I},\pi,\mathtt{t})}(es_2)$, the conclusion is valid.

All the inference rules used in the TRS are sound, therefore the TRS is sound. □

C EXECUTE THE LOOP BODY TWICE TO COMPUTE THE INVARIANT

Theorem C.1 (twice). *The deterministic loop invariant can be fixed after the second run of the loop body.*

Proof. Let $\langle H, C\rangle$ be the initial program state, where H is the history trace, and C is the current time instance. Given any program P, we use H_P and C_P to denote the history trace and current time instance by executing P. Based on the semantics of synchronous programs, there are three possible kinds of loops:

Case 1: loop {Yield; P},
The first run: $\langle \epsilon, C\rangle$ Yield; P $\langle C \cdot H_P, C_P\rangle$
The second run: $\langle \epsilon, C_P\rangle$ Yield; P $\langle C_P \cdot H_P, C_P\rangle$
The final effects: $H \cdot C \cdot H_P \cdot (C_P \cdot H_P)^\star$

Case 2: loop {P; Yield},
The first run: $\langle \epsilon, C\rangle$ P; Yield $\langle (C||H_P) \cdot C_P, \{\}\rangle$
The second run: $\langle \epsilon, \{\}\rangle$ P; Yield $\langle H_P \cdot C_P, \{\}\rangle$
The final effects: $H \cdot (C||H_P) \cdot C_P \cdot (H_P \cdot C_P)^\star$

Case 3: loop {P1; Yield; P2},
The first run: $\langle \epsilon, C\rangle$ P1; Yield; P2 $\langle (C||(H_{P1} \cdot C_{P1})) \cdot H_{P2}, C_{P2}\rangle$
The second run: $\langle \epsilon, C_{P2}\rangle$ P1; Yield; P2 $\langle (C_{P2}||(H_{P1} \cdot C_{P1})) \cdot H_{P2}, C_{P2}\rangle$
The final effects: $H \cdot (C||(H_{P1} \cdot C_{P1})) \cdot H_{P2} \cdot ((C_{P2}||(H_{P1} \cdot C_{P1})) \cdot H_{P2})^\star$

In all possible cases, the loop invariant can be fixed by the end of the second run. (cf. Fig. 1.) □